Software Complexity: Towards a Distributed Governance of a Production System

Evgeny Vasiliev

Thesis for the degree of Doctor of Philosophy submitted to the Department of Media, Communications and Cultural Studies, Goldsmiths, University of London

June 2023

Declaration

I declare the following thesis to be my own work. Where the works of others are used, they are cited and referenced in the bibliography. Any assistance from others is listed in the acknowledgements.

Candidate Name: Evgeny Vasiliev

Student Number: 33514675

Date: 30/06/23

Candidate Signature:

Acknowledgements

I could not have done this work without the tireless care and support of many individuals, both in and out of my research pursuits. Most importantly, my two supervisors Marina Vishmidt and Graham Harwood, as well as Matthew Fuller, who would always be there with invaluable advice throughout the key milestones of the project. I admire their commitment, trust in what am I trying to do and the inspiration they continued to provide throughout the years. Also, my appreciation goes to many other members of the Cultural Studies and Sociology departments at Goldsmiths for their feedback and support during my time there, particularly Brian Allende, Sean Cubitt, Luciana Parisi, Shela Sheikh and Gareth Stanton. In terms of fieldwork, I'm indebted to Juan Settecase for his enthusiasm for DevOps and patience when explaining it to me. To Joy Mariama Smith, for the inspiring discussions around the possibilities of queer performativity. To Alejandro Cerón, for providing an opportunity for a research period in the Netherlands, as well as stimulating theory discussions over breakfasts. To Maria Beatrice Giovanardi, for the emotional support throughout the intense writing time in Indonesia.

To my work colleagues at the two organisations that I was lucky enough to work with throughout my research, who at various times took part in the lively and fruitful discussions about the various aspects of my professional and theoretical engagements with software production. Wherever you are, I send you my warm greetings, and looking forward to working with all of you again.

To all the other friends and colleagues for being my interlocutors and for providing direct and indirect support, including Daniil Alexandrov, Jennifer Chater, Dimitra Gkitsa, Mo Jaeckel, Aaron Juneau, Matthias Kispert, Conrad Moriarty-Cole, Henrike Neuhaus, Lennaart Van Oldenborgh, James Phillips, Panagiota Polychroni, Renan Porto and Hayato Takahashi.

Through my thesis, this extraordinary group of individuals comes together as a team that makes this particular way of thinking about software possible. I submit this PhD with the utmost interest in where this trajectory is going to take us next.

Abstract

This PhD is interested in the complexity which arises in software production due to the divergencies of the organisational and market forces and aims to find out what are the existing or potential new ways of mitigating the complexity effects. Viewing the distributed kind of governance characteristic for most complex systems together with the value produced, I discover that the complexity is often caused by *software capitalism* – a valorisation regime which perceives every firm primarily as a technology firm and creates profits by keeping its software in a state of perpetual disrepair. The software in such a regime acts as an interface between the domain of market exchange and the sphere of organisational culture, with the market tending to increase complexity so that the production intensifies, and the organisation resisting this tendency to be able to optimise the audit of the production performance. I approach the study from the standpoint of development operations (DevOps) to reveal the mobilisation of the software system's epistemology as a productive assemblage for planning and control, which becomes a key dynamic in the situation of uncertainty that complexity presents.

Coming from the experience of empirical work as a digital product lead, I try to view DevOps with diffractive and compositional optics, by explaining a software production system through the notions of the problem space and the epistemic infrastructure. This makes it possible to clarify the performative role of a software system in the capitalist mode of production as simultaneously a service and a product. While the software system is at any moment too complex to be fully repaired, its dysfunctional condition is further aggravated by the phenomenon of the falling cost of computation. Acknowledging this trend, I argue that mitigation methods should be investigated in a diffractive way, effectively attempting a queer methodology for DevOps, to open a conversation about a more-than-human representation of the space of production that sees the negotiation of shared meanings topologically and immanently, rather than based on the dominant hierarchies, pre-existing assumptions or external evaluation.

Table of contents

List of Figures

General introduction

The present study turns to the theme of technical complexity in software systems because it wants to find out why such complexity is usually mitigated by expanding production, which tends to make the system more complex, instead of troubleshooting existing technologies and improving maintenance, testing and support routines. In other words, why is there a tendency to make the work of production teams harder, rather than easier? The project is undertaken with the hope that the findings will give other researchers and practitioners an idea of what kinds of tools and methods can be used to minimise or avoid the detrimental impact of complexity effects. The project's central claim is that technical complexity emerges as a conflictual force between the spheres of market and organisation, appearing as a benefit for the former, and specifically for the circulation of capital, but is harmful to the latter, specifically to the organisation's operations. Over the years of my professional involvement with digital production, I couldn't help but notice that the software and operations technologies that are available to present-day organisations are often underutilised or ignored, and my concerns were growing with regards to what kinds of risks this may create for long-term software system maintenance. Rather than being designed according to the demands of actual production teams and the larger goals that brought them about in the first place, the software systems, I suspect, more often than not are amended to conform to the constraints of what would seem an unnecessary bureaucracy. On a larger scale, my argument became that any technical complexity that may exist in society's software systems, for whatever reason, appears as a barrier to the optimal functioning of society's vital infrastructures, and has to be lessened, alleviated or otherwise diminished in its effects.

At the same time, continuing with my research, I came to acknowledge that even though the tensions within the space of production intensify the system's complexity, the complete eradication of complexity is not a goal in itself, since some of the complexity present in the system is an essential as a part of its specification, and after some point decreasing the complexity is not possible without restricting the system's functionality. Furthermore, some of the complexity avoidance can be dealt with at a policy level by shifting the mode of governance from centralised to distributed or achieving a balance between the two. As I compare the industrial mass manufacturing model to software production throughout the thesis, I discover that in a factory context of a relative scarcity of means, the governance achieves the best results by evolving in a centralised way. In high-complexity types of production, such as that of software, fast speed in conjunction with centrally oriented governance makes it

harder to meet the escalating uncertain conditions and creates delays which may, to varying degrees, impede the required system adjustments. This, in turn, risks placing production teams under additional stress. Distributed governance, on the contrary, is better suited for absorbing the complexity fluctuations in that it allows the software system to evolve through its local relationships, by connecting their constitutive elements, using the specifications as a guide, and yet constantly renegotiating them because of the changing context. Yet, as I find towards the end of my inquiry, the distributed model cannot fully address my concerns in that it does not reduce the complexity, but on the contrary, makes it possible to create more of it for further capital circulation.

Which complexity?

To create the context for the five chapters of my thesis, this General Introduction is split into three sections. The first section outlines the main parts of the argument. This includes explaining the specific meaning of high and low-complexity production my research investigates, the value and control axes it uses for navigation and the meaning of valorisation it identifies in the software complexity context. The section concludes by explaining why the research primarily engages with development operations (DevOps) and takes note of Agile and waterfall as the two key production paradigms that the project as a whole relies on. The second section briefly introduces my empirical research. The third section lists the main thesis definitions using a lexicon style, with a plan to provide more detailed commentary in the later chapters. Lastly, the Introduction provides chapter overviews and ends with an overall summary of the argument.

High and low complexity production

While much of the research around the present conjuncture of labour and management in production comes under such banners as *digital labour* or *full automation*, it is necessary to begin by explaining why the motivation in my project is different and in which way I intend to combine their findings with those of management studies. The goal of digital labour research is generally understood as developing a critique of the rationale for labour underpinned by the value relation. This paradigm had initially been posed by classical political economy and formalised by Karl Marx and was later enacted in the industrial research of such schools of thought as Frederick Winslow Taylor's scientific management and human relations movement pioneered by Elton Mayo. More recently, the critique has informed the programme of critical management studies and is indispensable for critical analysis of many facets of the contemporary manifestation of capitalism, including but not limited to Silicon Valley ideology, mineral extraction, or the overexploitation of workers stripped of their basic rights on many digital platforms. The research trajectories in this camp often assume that the value relation is the key underlying problem of current social formation, and often employ critical analysis with the aim of proposing

an exit strategy from such relation. Yet, due to the broad meaning of the term *digital labour* and its predominant involvement with value relation, in my research, I'm leaning more towards the term *software production*, which also enables me to orient the discussion towards DevOps and complexity challenges.

Furthermore, using software production as the main reference makes it possible to differentiate the roles of the participants of the two general varieties of production this thesis works with: software production, and the mass manufacturing of physical objects. I see the latter as a low-complexity production due to the finite nature of the constituent parts of inventory and machinery on the factory floor. The former, due to the computational nature of the parts and tools, on the contrary, has the potential for a largely unrestrained entanglement of interrelations and layering of software abstractions. This precludes any parsimonious explanation of the system and creates high-complexity production situations. Leaving the rich discussion of material and immaterial production out of the scope of the present thesis, there is an additional important upshot to this complexity dynamic, which can be identified as the correlation between the empirical – or in fact, ontological – and the epistemic facets of the social classes in the way they are enacted in the production process. Empirically, there is a split between the working class and the capitalists, where the workers embody the productive function and stand in antagonistic relation to those who own the business and act as the force that alienates the workers from the products of their labour. In terms of epistemology, software is ambiguously present as a service or a product at the same time. Where it appears as a service, such as in maintenance, support or integration efforts, there is no easier way to alienate the workers from the labour outcomes than in any other service work, since the product of the service, to use Marx's theorisation, cannot be separated from the act of producing.[1] For business owners, however, software products are present as commodities available for exchange in the market, for example, in the form of specific product releases. Such correlation between the roles and manifestations of software means that even though the software is available as a product to users and business owners, it is still conditioned at every step on the service work done by the production team.

The value-control axes

This is where the present study sees the importance of connecting the digital labour critique with complexity management studies – the area of management research which engages with complexity sciences. In essence, the association of the empirical split with high-complexity production implies that anyone approaching software complexity in terms of the value present in its end products, and developing a critical inquiry by such means as industrial manufacturing operations research, will only have

---

[1] Marx, 1990: 1048.

access to a specific interpretation of the problem, for example, that capital can see software complexity as the pathway for improving capital circulation, through such means as expanding production, with such consequences as platformisation or casualisation of labour. What is equally important to understand, however, is that the software system evolves in the context of organisational dynamics, where the activities are not determined by the value in the economic sense, but rather by the shared needs, that is, they are present as the labour not directly mediated by the market. For example, such is the proposition for symmetrical treatment of firms and markets by the 20th-century British economist Ronald Coase. According to this interpretation, where the cost of price regulation is too high, the processes get internalised, thereby creating the space for developing the organisational theory.

In this context, the split happens organically, since the discussion of market relations is irrelevant within the organisation domain, yet the difficulty lies in the fact that neither one can be completely separated from the other. For example, as Chapter 3 explains, even where the circulation of value within the organisation may not be made explicit, the circulation of knowledge, which is often treated similarly in stakeholder relations, is clearly expressed in production at all times. In the context of the overarching market realm, the organisation domain exists as the intrinsic cultural sphere of interpersonal relationships based on tacit knowledge and practices. The value exchange here is voided to make the job easier for the accounting department, which, along with sales, takes care of maintaining the boundary between the events of production from the extrinsic demands of the market. Due to my interest in operations, however, the aim is not to try and pack the issues of value uniquely into the purview of the market and the issues of control into one of organisation, but to imagine value and control as two axes co-existing on the plane of the capitalist mode of production. The duty of operations in this case would largely be to balance the issues of capital circulation with the issues of control over such circulation.

Bringing value and control in as two axes could make it possible to reach beyond the critical paradigm described above. The complex software system would no longer uniquely belong to the facet of value relation, and thus there would be no pressure of finding the exit from capitalism or any other regime oriented towards value extraction. It becomes possible to focus on gaining a better understanding of what exactly is wrong with the existing methods of creating and maintaining software systems, and what is the specific way in which the sphere of value comes to interact with forces of administration in a production context radical in its variability and uncertainty. In other words, the iterative application of digital labour critique and management studies enables the comparison of the complexity impacts across both the spheres of circulation, of capital in the market, and of knowledge within organisational culture. In terms of research goals, creating a value-control frame makes it unnecessary to think about overcoming any regime organised by neither value nor control relations, and instead makes it

possible to portray a historically specific entanglement of the market and the organisation that leads to complexity spikes and ultimately to software crises.

Simultaneously, the value-control axes allow identifying the changing role of the organisational culture across the high and low-complexity production contexts. The former enjoys an added benefit of co-operation which, as Marx observes, emerges when the individuals partake in a collective endeavour, during which they activate their species' capacity to be able to effectively work in a planned way with others.[2] The same cooperative benefit is unusual for software production, where bringing new engineers to software projects does not add to the speed of delivery, but, on the contrary, is present as a burden to the already existing team in the initial stage due to the difficulty of gaining shared understanding in a highly complex environment. However, as Chapter 5 explains further, the longer team members collaborate within specific production conditions, either around a software system or within the organisation, the more accustomed they become to each other's working styles, creating momentum through making use of the tacit knowledge which accumulates within their practices. The layer of production culture that emerges from the entanglement of the tacit knowledges and local practices permits the stakeholders, as a unified interest group, to navigate the uncertain and inconsistent situation created by the local activities of individual agents. It acts as a vital organisational adhesive, providing the means of articulating problems in teamwork, and requires the agents to engage with unproductive labour – the activity that does not carry the surplus value or product for market exchange as its immediate outcome. The kind of labour that is concerned uniquely with the culture of production within an organisation often proves to be so important as to necessitate introducing various sorts of production management staff, such as delivery managers, information architects or business analysts, who dedicate their efforts specifically to meaning-making.

Towards valorisation in software capitalism

Having established the coordinates of value and control to orient itself within the research terrain, the argument inquires into the problematics of valorisation, or specifically the strategies that the software capitalist mode of production employs to be able to continue reproducing itself. Having seen that the market domain deals with a notion of value which is different from the one of the organisation and approaches the technical system in terms of its exchange value capacity, it may be possible to suggest that there is a way for capital to convert the hindrance of technical complexity to the means of generating profits. The conversion here should be achievable in the fashion theorised in Marx, through reducing the value of necessary labour time: 'everything that shortens the necessary labour-time required for the

---

[2] Marx, 1990: 447.

reproduction of labour-power, extends the domain of surplus labour.'[3] In the case of software production that deals with information as goods which do not perish or reduce when consumed is best achieved through expanding production, since such expansion, as the thesis later notices in terms of the notions of fractality of production system design and the falling cost of computation can be done without significantly increasing the necessary labour time. Notably, it is also possible at a low cost, since some of the real-world constraints are no longer present in digital production as compared to the manufacturing of physical objects.

For example, there is often no clear-cut divide between production and distribution. The study of logistics, such as the one offered by the sociologists Sandro Mezzadra and Brett Neilson, clarifies the involvement of Marxian politics of distribution in present-day supply chains. To Marx, who never used the term *logistics*, transportation appears as the extension of production into the circulation – 'continuation of a production process within the circulation process and for the circulation process'[4] As Mezzadra and Neilson observe, transportation is not a mere means of reducing the costs of commodities that have the surplus value fixed at the moment of their production. Instead, the surplus value changes depending on the mode of logistical coordination. Logistics are the means of regulating the turnover time, typically to make the acquisition and move the inventory in the fastest possible way, which increases the gains by shortening the transition time during which the capital cannot convert the surplus value into profit.[5] Transportation in DevOps usually appears in the form of a dashboard or another interface solution that makes it possible for the operations to manipulate their logistical resources on the cloud computing platform, such as Amazon Web Services (AWS), which I have encountered in this capacity as part of my fieldwork. The resources offered by cloud operations consist of radically different technologies and paradigms and come in such forms as databases, virtual servers or message brokers, which often require a variety of professionals knowledgeable in their specific areas. In the final instance, the platform interface serves as a means for interoperability between the organisation's technology and the market, where the technology is delivered through symbolic manipulation. In this sense, DevOps takes production and distribution together without having to resolve the ambivalence of relations between the two. This, however, is precisely why my thesis is interested in DevOps, rather than any other area of software production uniquely belonging to either its production or distribution aspects.

---

3 Marx, 1990: 470.

4 Marx, 1992: 229.

5 Mezzadra and Neilson, 2019: 150.

DevOps and knowledge work

To explain what my research means by DevOps, it is important to sketch the general context that I assume here, frequently referred to as *knowledge society*. A knowledge society can be defined as a society where the factors of value and wealth creation prevalent in the era of the industrial mode of production – capital, labour and natural resources – have another major addition in the form of knowledge. In the dominant economic definition provided by management theorist Peter Drucker, *knowledge society* refers to a society where it is no longer capital nor natural resources, but knowledge which is present as the most important wealth-creating factor.[6] Furthermore, according to the observation of the historian James R. Beniger, there is a long-term trend of the increasing share of the knowledge constituent in the overall labour force composition in the Western economy. While in the hundred years leading up to the end of World War 2 employment in industrial manufacturing dominated the market, it was on the steady decline after that point in favour of information-based work. [7] The present thesis, dealing with more recent interpretations of the knowledge-based business models from mid- 2010s onwards, sees the information technology (IT) sector knowledge as coincident with business value to such an extent that it becomes possible to write down the business strategy in the form of computer code. This makes the model different from industrial mass manufacturing of the pre-software age, where the implementation of the work process on the shop floor had been removed from sales, marketing and distribution.

In this context, *DevOps* is a relatively new methodology concerned with the production of the means of production and is located at the intersection of development and information technology (IT) operations. Such a location implies that DevOps deals closely with how knowledge is created and circulated throughout software systems, in which sense it is a methodology tightly interweaved in the knowledge society fabric. DevOps emerged as a result of the frustrations that had accumulated in the first decade of the 2000s when the increasing speed of delivery of production teams was systematically held back by the business operations, who often worked separately from development and therefore could not follow their continuously changing requirements closely enough. The organisers of the first DevOps conference held in 2009 argued that it is possible to converge the two disciplines of development and operations to be able to deploy and integrate new software releases in a seamless fashion, which would help alleviate some of the software production complexity. For the present research, the three aspects of DevOps are particularly important.

---

6 Drucker, 1993: 7.

7 Beniger, 1986: 23–24.

First, since software systems tend to expand, organisations are faced with the necessity and, simultaneously, the difficulty in onboarding new team members. In knowledge work, the specific character of entanglement of each job role may be different even if the set of skills required to do it is the same, due to the complex nature of products. In addition, there is a difficulty in onboarding new team members and developing the web of implicit or tacit knowledges, as we saw above, which creates additional pressure on the firm's communication capacity to keep pace with the growing population of its engineering teams. In his foundational work on software production management, Frederick Brooks had signalled an early warning that even a linear increase in staff creates an exponential increase in communication pressures related to training, onboarding and socialisation into the organisational culture.[8] Furthermore, it is not only the coordination of work which is communicative but the professional practice itself. While tacit knowledge cannot be easily communicated across the whole production space, DevOps aims at solving this problem by expanding communication bandwidth between development and operations and making the knowledge explicit.

Second, DevOps plays a significant role in mitigating the effects of backward incompatibility, or regression analysis, across the software system as a functioning entity. This effect is also present in traditional technological systems as reverse salients or older components can render innovation useless, but in software negative consequences may have an immediate impact across the whole system, depending on the nature of legacy code. Needless to say, the situation is rich with organisational particularities as to why specific old and new components are used and sees DevOps working with the organisation's policies, contractual agreements and personal idiosyncrasies that the technological system comes with. The third facet is organisation design – while DevOps may be involved with this activity in a greater or lesser capacity, the common understanding of the production goal implies that all of the parts in the production space are functioning to make it easier for the teams to meet the software system's requirements. One of the ways to do this is through systematic efforts at negotiating the design of the organisation towards the intended design of the system it works with. Such organisational flexibility becomes possible in the context of the Agile method of production, which takes precedence over the waterfall style when it comes to performing day-to-day software development.

Waterfall workflow in software production

While the Agile methodology will be more fully addressed in connection to the incremental delivery discussion in Chapter 1, at this moment it is necessary to note the distinctive features of the waterfall style of production, prevalent at the peak of the industrial era, that changed the specificity of its ap-

---

[8] Brooks, 1995: 18.

plication in complex production scenarios. As management theorists Arash Azadegan and Kevin Dooley explain, the waterfall is a project management paradigm that assumes that any production effort has to be executed only after it is clear what work needs to be done, and how and by which staff. For example, each design effort needs to start by identifying the requirements, linking them to the design activities, and finally, carrying out design and integration work. The paradigm came to be known as *waterfall* because the approach is strictly linear, making it easy to swim downstream, but harder to go upstream if any of the earlier activities have to be redone.[9] Such inflexibility makes the approach less fit for situations where customer requirements are likely to change, or for risky innovation cases, where the viability of the whole endeavour can only be confirmed after substantial design work has been done upfront. Vice versa, the waterfall is highly effective in the assembly line style of mass manufacturing, where it enables accommodating technical complexity by organising a finite number of operations around a finite number of components to produce large quantities of concrete immutable artefacts. In other words, since waterfall achieves its effectiveness due to the ability to account for all the possibilities and complexities it may contain, it does not apply in the same way in the production of software systems, since there is no certainty around the system's complexity. Four key reasons cause such uncertainty, which I should now briefly address.

First, unlike in industrial manufacturing, the components are often themselves software systems and can change during production – for example, security or database protocols developed by third parties or in open-source communities. Second, the software system tends toward a structure that doesn't have any repeatable components – a principle standing in diametric opposition to industrial mass production, where the repeatable contents, such as car parts, are the key to the functioning of the assembly line. As Frederick Brooks points out, in software systems 'no two parts are alike' because, according to the best engineering practice, every repeatable part should be replaced by a subroutine or a single component that can be called multiple times.[10] This means that any effort of scaling the system is not a simple operation of repeating the same elements on a larger scale, but a qualitative change of the system. Moreover, elements interact irregularly and have numerous states, or unique sets of attributes, which change over time. This creates an additional difficulty in reproducing the conditions for testing. Third, software engineers, unlike factory workers, perform operations such as finding creative solutions to problems and evaluation of existing code – not to mention the matters of many other IT occupations, such as project managers and business analysts, who deal exclusively with the creation of organisational culture and with the negotiations in the problem space of production. All of these types of

---

[9] Azadegan and Dooley in Allen et al., 2011: 427.

[10] Brooks, 1995: 182.

work are hard to break into sequences of established operations to create an assembly-line style production. Fourth is the so-called Moore's Law, a claim that the capabilities of hardware tend to exponentially increase in capacity over time. A co-founder of Intel, one of Silicon Valley's leading semiconductor manufacturers, Gordon Moore, proposed this principle back in 1965, which since then has proven to be one of the more persistent industry trends. The increase in hardware capacity creates uncertainty around the costs of production, and indeed, the costs of computation itself. The consequences of the falling costs of computation will be addressed more fully in Chapter 5, while Chapter 3 explains, via the operations research of production analysts Stefan Thomke and Donald Reinertsen, what the risks are of carrying over the assumptions of mass manufacturing to the production of software artefacts.[11]

The above reasons introduce a new sense of a disruption-based production which necessitates the use of more flexible just-in-time, continuous, or other Agile techniques, that make it possible to renegotiate when and what exactly is being delivered. This, however, does not mean abandoning the waterfall style of planning entirely, since the waterfall continues to prove more reliable than Agile when balancing multiple stakes across different departments. For example, in my empirical research, the waterfall style was necessary when formulating the general production stages, as shown in Fig. 2, coming from delivery through design to testing and development. The waterfall is also applicable to larger strategic initiatives, such as programme and portfolio management, where the roadmap sequencing of processes is important, as it acts to simplify the coordination of the company-wide production efforts.

Introducing empirical work

The case studies are based on my previous employment at a company referred to throughout as JX, where my product management experience largely comes from. JX is an online media outlet publishing daily briefings on a variety of cultural topics, focusing on young creatives. Throughout the years, my duties at JX have shifted from designing the product in terms of its visual aesthetics to designing the team relations in my capacity as a digital product lead. This was due to my growing practical experience, the progress of my PhD research and the deepening familiarity with the organisation's institutional ecology. Having supported the organisation during its transformation to a fully online publishing model in 2018 and throughout the switch to remote operation during the COVID-19 pandemic in 2020, I was simultaneously involved with many different aspects of the organisation's software – a content management system (CMS) that the organisation had been running on.

---

[11] Thomke and Reinertsen, 2012.

While the digital publishing platform was something that developers and editorial staff were dealing with the most, my managerial activity was mostly connected to Atlassian Jira, a support ticket software, which came to the forefront of my interest because of its specific approach to knowledge. While also focused on writing, Jira appears different from traditional word processors because it is not a linear writing tool. Each issue or ticket created in it becomes a knowledge node of its own, allowing it to collect the information collaboratively by teams and later search and sort through it. In other words, Jira acts as the epistemological tool for organising the knowledge captured by the collective use of its tickets. Second, concurrently with the interests of my dissertation writing, my professional duties have shifted towards DevOps, in such aspects as infrastructure and security issues which the organisation was dealing with. While at first the work seemed exceedingly technical, upon further investigation it appeared rather as a concise representation of business operations, written down in the lines of code. Beyond informing my research writing, the more hands-on involvement with the software system made me think more generally about the formalisation of the technology-positive discourse across the organisation as a whole. This had organically led me to think topologically, both about production workflow via the stream-aligned DevOps paradigm, and about organisational culture, in terms of its engagement with the software and the communities of practice. Throughout the casework, the application of the topological method made it easier to emphasise the continuities between the organisation's different departments, which helped production, editorial and executive teams to discover that many things that previously seemed radically different have much in common.

Term definitions

Before starting the discussion of the thesis's main argument, it is worth clarifying some of its core notions, since they are frequently used in various sources and their meaning may vary. The section provides a quick guide and plays the role of a lexicon, sketching out the contours of the terms without going into much detail, which is provided later in the thesis.

System

The thesis generally adopts the widespread systems approach to production and defines the system as a whole that functions by the interaction of its parts. The two types of systems are distinguished as centralised and distributed. The former type is hierarchical, less flexible, yet capable of developing great production speed when no change to the process is necessary. The latter type is complex and agent-based. The systems of this type avoid confrontation with the complexity of the software, yet may present a difficulty for centralised control and are often governed to some degree in a distributed way. Moreover, *software* as a general term is distinguished, for present analysis, from *software system*. The former is usually used to refer to computation-based tools, applications and services in the general

sense. The notion of software as a system, however, aligns here with the Science and Technology Studies (STS) research programme in its focus on the specific relations that any technological system has with the society it exists in, shaping it and being shaped by it in return. The research frequently refers to the system under consideration as a software production system, which implies that it is not a user-facing piece of software that the research is interested in, but in the software system that is used by the production team – therefore the focus of research is shifted towards IT operations and how the process of production itself is made possible with software. Another important feature of systems presentation of technology is making it available for *audit*, defined as a control practice of reviewing and planning the production activities orthogonally to the system's complexity. Systems of traditional industrial manufacturing are seen as centralised, while software production systems are seen as too complex for centralisation and tend to be governed in a distributed way.

Traction. Some of the attributes of the system that this research works with are control, momentum and traction. Traction, for the present study, is used as a heuristic that estimates the effectiveness of a system's operation, and is expressed as a relation of change to control. As discussed in Chapter 1, complexity increase within a system is constant and necessary, therefore in traction, the rate of change instructs how fast such increase happens, while the notion of control, following the definition of the historian James Ralph Beniger, stands for the progression towards a goal within a system, conditioned by its communication means.[12] The loss of traction means that a given administrative mechanism is overwhelmed by the escalating changes and results in systemic crises. I see such crises as characteristic of the present software capitalist formation and refer to them throughout the thesis as software crises.

Momentum. Related to traction is the notion of momentum, introduced by the historian and sociologist Thomas P. Hughes as a tendency of a system to continue its development along the previously defined trajectory,[13] and adopted in the present research as a relation of the rate of change to the system's operational maturity. The fact that some systems acquire great momentum is not so much linked to their size but rather depends on the density of the local knowledges contained in the community of practice pertaining to the system, and the coincidence of the organisation's design to the design of the system. While momentum can be used to estimate the effectiveness of operations in the same way as traction, the latter term makes it possible to account more prominently for the involvement of the administrative mechanisms within production.

---

[12] Beniger, 1986: 8.

[13] Hughes in Bijker, Hughes and Pinch, 2012: 70.

Reverse salient. Also adopted from Hughes is the notion of reverse salient, which denotes a system's component which has fallen behind and out of phase of the overall operation. Reverse salients can occur in the system's technology, as well as in its social relations, and are important for balancing the system's traction. In terms of technology, reverse salients are present at every system update and are usually addressed by such procedures as regression testing, which verifies that the new source code does not conflict with the functioning of the existing application. In terms of the organisation's structure, the reverse salients may occur where the changes are dictated by innovation external to the system and cannot be supported by the organisational ecology. This is the case where the system's traction is stifled by its momentum and can be a cause of software crises.

Conceptual integrity. The notion of conceptual integrity is borrowed from Brooks, who interprets the software system as something that needs to be designed by one mind.[14] The notion is important for understanding the functioning of centralised systems, and therefore I refer to it throughout my thesis as a focal point which makes it possible to compare the various aspects of centrally-managed systems with the ones of distributed type.

Complexity

The phenomenon of *complexity* in software systems is defined rather narrowly for this research as the epistemic opacity in the relations between the components of a software system. Such definition focuses the thesis on the specific problem within operations research, rather than having it measure up to the broader discipline of complexity theory or its applications in computer science or management – which nevertheless are the two disciplines which, to a large extent, define the matters of the present research. The present specialised kind of complexity is close to the account of the sociologists of technology, Annemarie Mol and John Law, and necessarily deals with the things that relate but don't add up, with the events that occur but not as causes or effects of one another, and with the phenomena that share the same space but are difficult to map into one set of coordinates.[15]

Complex system. The notion of a *complex system* in my research relies on the definition of the management scholars Steve Maguire, Peter Allen and Bill McKelvey, to see it as a whole which is made of parts or agents, where such agents are conditioned by different forces that relate their behaviour in a specific moment of time contingently to the states of other agents, resulting in complexity as a phenomenon which presents patterns which are neither fully predictable nor fully random but rather can-

---

[14] Brooks, 1995: 255.

[15] Law and Mol in Law and Mol, 2002: 1.

not be explained in a parsimonious way.[16] Complex systems, in the view of Steve Maguire, usually have large numbers of dynamic components that interact in a non-linear and short-range fashion, have feedback loops because of such interactions, and are ignorant of each other's actions. Complex systems have histories and tend not to be balanced.[17] What is at stake here are the causes and relations between the parts: where the states of the parts in technological systems are stable and independent, devoid of causal relationships, the systems are seen as simple, and where the parts are interrelated, the systems are seen as complex. Previous studies of this kind of complexity largely relate to *self-organisation*. The latter term, in the explanation of the complexity scientist Melanie Mitchell, refers to the ability of large numbers of a system's simpler components to organise themselves without any central control, and being able, through such self-organisation, to use information, to learn and evolve.[18] In complexity management studies, complex systems can be present on different levels, depending on the matters of analysis: single organisations and business units, segments of value chains and decisions, or larger entities, such as industry sectors and whole economies.

High-complexity production. The thesis begins defining the process of production via Azadegan and Dooley, who see it as the creation of goods and services, or, more specifically, a set of interrelated operations aimed at transforming resources into outputs.[19] Surpassing such definition, however, I am interested in studying production beyond its outcomes and use the example of a software production system as a means of high-complexity production encompassing the organisational context and the relations between agents of various kinds. This enables me to fully grasp the outcomes together with the production of the means of production. Due to the focus on operations, the production of software here only concerns software systems that display the kind of complexity discussed above. Thus, the research is not concerned with auxiliary tools and services which are sometimes referred to as *middleware*: messaging systems or the application, database and web servers. Such services are tightly standardised and usually considered a part of a wider ecology of deployment, where they function as connective tissue between the operating system layer and the application layer. The focus here is instead on the higher-level entities, where it is not easy to follow the pre-selected templates. Moreover, the actual results of production are of lesser importance for the objectives of this research. Rather, the priority is to think about the production process in its connection to the problem space and its epistemology – how the knowledge is collected, organised and put to use throughout the process. Generalised in these terms, the research does not intend to uniquely apply to the production of software, but to production in gen-

---

16 Maguire, Allen and McKelvey in Allen et al., 2011: 2.

17 Maguire in Allen et al., 2011: 82.

18 Mitchell, 2009: 4.

19 Azadegan and Dooley in Allen et al., 2011: 418.

eral. This is possible not least because software at present constitutes, through the ubiquitous presence of computation in all of its subroutines, an indispensable part of any production process, including industrial mass manufacturing.

Complexity and open source. In the context of such an interpretation of high-complexity production, it should be mentioned why the present research is not directly dealing with any of the production-related debates involving open-source communities. The general reason is that the state of open source infrastructures and ethics is a rather particular case in terms of the market-organisation divide. The situation here is reversed: the market domain here is failing to valorise, while the organisation domain is suffering from the lack of operations staff. This peculiar problematic is discussed in more detail by the open source pioneer Eric Raymond and more recently in the industry reports such as the one created by the researcher and writer Nadia Eghbal for the Ford Foundation.[20] To Eghbal, the open source has no shortage of programmers willing to write new code, even though the labour is unpaid. Volunteer engagements are largely caused by the desire to strengthen one's position in job interviews and other reputation-related reasons, while a growing share of open-source contributions come from the industry giants like IBM, Microsoft or Intel.[21] Moreover, where the sponsorship is present, there exists a controversy about whether specific work is carried out because it is required or rather because of the available funding. Due to the same voluntary nature of this sector, however, the funding alone does not help tackle the main problem of open source, which is formulated as the lack of stewardship. This lack is largely manifest in the projects being under-resourced on many fronts not directly related to creating new source code, such as strategy, planning, code reviews and testing, technical documentation, as well as community advocacy and evangelism.[22] The problem of tackling software complexity in the sense of the present thesis is as relevant for large software systems produced by open-source communities as it is for large software systems produced by any other means. The present study, however, is mainly interested in the topological solutions to complexity, such as intentional organisation design, which means that the lack of stewardship and other problems that arise before the organisation design stage can be initiated lie outside of the scope of this research.

---

[20] Eghbal, 2016.

[21] An analysis of 2017 GitHub data conducted by business analyst Frank Nagle finds that some of the most active free and open-source software (FOSS) developers contributed to projects under their Microsoft, Google, IBM, or Intel employee email addresses (Nagle et al., 2022: 58).

[22] Eghbal, 2016: 125.

Market and organisation

While acknowledging that the domains of market and organisation are deeply intertwined and co-exist in the real-life production context as mutually conditioning spheres of relations, it is necessary to analytically split them for the present research to clarify the complexity effects. In a more strict sense, however, what I here refer to as *market* and *organisation* is close to the two kinds of mediation discussed by the Marxist feminist theorists Maya Andrea Gonzalez and Jeanne Neton as the spheres of production directly and indirectly mediated by the market. Gonzalez and Neton come from the idea that a common feature of all labour is that it has to be validated as such within a society, regarding a specific function it carries out within the process of production. Once socially recognised, labour can take the form of activities either explicitly productive and paid, such as manufacturing or service labour, or as more hidden and unpaid, such as domestic or care work. This, they argue, necessitates the split of labour into the *directly market-mediated* and the *indirectly market-mediated* spheres, which are more appropriate for identifying the role of labour in the production process than other distinctions – such as productive and reproductive spheres – because they are unspecific to what kind of activity it is or its use value. In the directly market-mediated sphere, the return on investment is primary. Therefore, the activities have to 'meet or exceed the going rate of exploitation and/or profit' while in the latter they are 'highly variable in terms of the necessary utilisation of time, money and raw materials.'[23]

Aligning with the division of market mediations proposed by Neton and Gonzalez, I focus the market mediation split, in terms of my interest in operations, on the character of exchanges among the individuals and collectivities. The directly market-mediated sphere, or simply *market* in the present research, deals predominantly with value exchanges, such as of commodities produced independently by otherwise unrelated entities – either individuals or firms. The indirectly market-mediated sphere pertains to exchanges that do not involve exchange value as its primary concern but are determined by common needs and purposes, often in the form of security or compliance demands. I categorise these as pertaining to the *organisation* – either the state or any other institution that operates as an administrative body based on the social contract, rather than value exchange between independent entities. Such a distinction makes it possible to clarify the different aims the two domains pursue: the market mediation is focused on the labour outcomes, while the organisation is concerned with the process of production, which is entangled within the organisation's discourse and cultural norms.

Business and the technology value streams. The professional DevOps literature implies a specific intra-organisational kind of value expressed in the effectiveness of operations which is closely re-

---

[23] Neton and Gonzalez in Endnotes 3, 2013: 63.

lated to the stream, or a continuous flow of work tethered to a business domain or organisational capability.[24] The two facets of the stream are frequently evoked as the business and the technology value streams. As the DevOps practitioner Gene Kim explains, the business value stream is more general and defines a series of any actions aimed at fulfilling the customer's requirements, while the technology value stream is a series of steps that aim to deliver the customer request through transforming a business idea into a technology-driven service.[25] As Chapter 3 discusses further, the technology value stream usually utilises a *deployment pipeline* – a production technology that automates the delivery of software products to the stakeholders, and which is usually kept coincident with the business value stream.

Production design lifecycle

The thesis works towards a model, fully presented in Chapters 3 and 4, of the production process activated by the circulation of knowledge between the *epistemic infrastructure as code* (EIAC) and the *problem space of production*. The former is the infrastructure that describes the ways of knowing about the software being produced, and the latter is the space where the problems are negotiated between the stakeholders. The two parts are linked by deployment and integration. *Deployment* is a set of processes that make the system available to the stakeholders. *Integration* is the assimilation of knowledge, understood here as incorporating the results of problem negotiations back into the epistemic infrastructure so that they become a part of the overall body of knowledge about the system. The deployment deals with the four main types of system components described in the EIAC – the data, the host environment, the configuration, and the source code, which are the conceptual devices that provide a new layer of abstraction in the deployment. Such abstraction makes it possible for DevOps to address the complexity that arises from the issues of compatibility between the components separately from the writing of code, testing and other production activities. The production as a whole can be referred to as the design lifecycle because all of its parts co-exist in the process of continuous readjustment, with each iteration instructing particular details of the cycle's design. The system traction metric can be applied in the production cycle to understand whether the EIAC and the problem space adhere well enough to ensure smooth and seamless circulation.

Once the product is deployed, it becomes available to the stakeholders, who take part in the negotiations within the problem space of production. The thesis accounts for the three criteria. *Requirements* are the descriptions of task-specific givens, goals and operators. *Acceptance criteria* are the definitions of done, which are used to understand if the customer value has been delivered. The criterion of *customer*

---

[24] Skelton and Pais, 2019: Ch.5.

[25] Kim et al., 2016: 7–8.

value, in turn, is used to define the priority of task completion. The materials that circulate in the production system cycle are referred to in the present thesis as knowledge *inventory*. Such inventory can be of at least two general varieties. In deployment, inventory is the software components which are assembled into the product, whereas in integration, the inventory is the new knowledge about the system, and it becomes a part of the production system's knowledge base in such forms as technical documentation or policies.

Agents and stakeholders

For the present research, the terms *agent* and *stakeholder* are close in that both refer to the individual entities that activate the production lifecycle, with a key difference that agents can be any kind of entity, human as well as non-human, while the stakeholder category here denotes business owners, production personnel and product users, all of which are assumed to be specifically human individuals or collectives. The use of either one of the terms is chosen based on the aspect of the context of their involvement with the production process. Notably, the terms differ from the category of social class in that neither stakeholder nor agent implies hierarchical distinctions or any sort of antagonism in interrelations. In terms of its formal definition, the term *stakeholders* is used to refer to the groups that could be potentially affected by the changes in production. The notion of *agent* is viewed here through the lens of complexity science, where it implies autonomy and self-organisation, and can refer to any entity which has some sort of agency. According to the definition of the computational neuroscientist Péter Érdi, agents are 'autonomous computational units endowed with the authority for decision-making or strategic decisions or … to make selections among possible strategies.'[26] The management scholar Max Boisot observes, in addition, that agents are characterised by their states, which change depending on their interactions with other agents. Such interactions are usually regulated, and can show the agents form associations either locally with their neighbours, or globally.[27] Due to the agents' capacity to act both in larger assemblies and independently, their function can be approached in two ways. On the one hand, the agents are the participants of self-organising processes that see them enter into temporary associations to address specific cases of complexity. In this sense, the agents should be approached in terms of the effectiveness of their collective endeavours, which can be evaluated through such mechanisms as audit. On the other hand, agents are individual entities largely involved with the assimilation of knowledge, in which respect the system's resilience to complexity effects is best evaluated through such individual agent characteristics as cognitive load.

---

[26] Èrdi, 2008: 307.

[27] Boisot in Boisot et al., 2007: 8.

Chapter overviews

Given these main themes, the thesis is structured in the following way. Chapter 1 reviews the project's core literature sources by comparing the major foundational texts of the past with more recent thinking. This enables me to trace the changes that have emerged in the field since the primary software operations research was done. The first section looks at the analysis of technology in terms of systems as theorised by Hughes. The key problem here, pointed out by Hughes himself, is the control crises that such systems appear to have as their recurring attribute. I turn, for the possible solutions, to some of the recent complexity management theories that expand the systems thinking via the application of techniques of distributed control. The second section examines the seminal work of Frederick Brooks. The core problem that demands attention here is that any organisational dysfunction bears a direct impact on production. The response is found in the DevOps stream alignment organisation design paradigm, based on the theory of Brooks's contemporary, organisation theorist Melvin Conway. Stream alignment suggests the intentional organisation design that conforms to the desired topology of the software system.

The third section of the literature review outlines the context of the research as *software capitalism,* a term adopted from the organisational scholar Nigel Thrift's notion of soft, or knowledge-based capitalism. The section establishes the grounds for a more substantial discussion of software capitalism in Chapter 4 by explaining the three shifts that had historically accompanied the soft capitalist transition. These are the shifts in epistemology, technology and methodology. Epistemological shift prohibits the strategies' reliance on the consistency and stability of outcomes and prescribes instead planning to be done with uncertainty in mind. Technologically, the new computational capacities increase the communication bandwidth and make the mastery of acquisition and assimilation of knowledge a competitive feature of firms in the market domain. In terms of methodology, this means a growing interdisciplinarity of production approaches, with DevOps being one such example. Alongside the three shifts, one overarching principle of software capitalism is seen as the coincidence of the business value stream and the deployment pipeline of the software system the business is running on. As the DevOps practitioner and author Michael T. Nygard has it, 'there's no such thing as a website project. Every one is really an enterprise integration project with an HTML interface.'[28] The key importance of soft capitalist context, which is also featured prominently in Thrift's work, is the notion of *culture*, viewed here more specifically as the culture of software production within an organisational space.

---

[28] Nygard, 2007: 278.

Chapter 2 describes the research method through the three problems that it has to address: the heterogeneous matters of investigation that cross the disciplinary borders between cultural and industrial research, the uncertainty of the domain, and the study's inevitable empirical grounding. In the first section, the interdisciplinarity is engaged through adopting a *compositional methodology* (CM) framework, understood here as the study of the possibilities that a problem has in the process of its involvement with resolutions. The section introduces the two general categories of CM at stake here: the epistemic infrastructure and the problem space.[29] The former of the two is explained as the suite of principles for organising what is known about the problem, while the latter is the domain where the possibility of the problem exists. The problem is thus not something that is given, but instead emergent in the process of the application of the method. Second, the uncertainty of the domain is addressed by bringing together the compositional methodology, via the cultural theorist Celia Lury, diffraction, via Karen Barad, and abduction, via the philosopher and logician Charles Sanders Peirce as well as the post-Peircean philosophers of science such as Lorenzo Magnani.

The compositional method is here seen as topological because it concerns the study of matters in terms of their proximities and borders, and builds upon the similarity of the methods in cultural studies and the topological treatment of operations in DevOps. The cultural topological method is understood in terms of its view of software as a tool for the new formalism, which extends reasoning beyond the limits of senses or bodies. Instead, such formalism replaces the actual relations with their quantitative representations, such as indexes, metamodels or networks. These representations are then analysed in terms of their topology, relying on mathematical principles of the study of space, which allows the theorists to describe more specifically the relations between particular cultural phenomena and their changes over time.[30] The second facet, team topology, is a set of principles for developing a sense of strategic awareness of the evolving software system. To use the terms of the team topology paradigm creator, Matthew Skelton, its principles are technology-agnostic and allow flexibility to adjust the team configurations depending on the operational requirements.[31] The research aims to bring together these two elements of the method by shifting the focus from the culture in general to specifically the culture of software production embedded within an organisational sphere of norms and values.

The method of diffraction affords the necessary degree of precision when a more nuanced dealing with case studies is required. The production process is viewed as disruption-oriented, which means

---

[29] Lury, 2021: 141.

[30] Cf. Lury et al., 2012.

[31] Skelton and Pais, 2019: Glossary.

that the problems are described through discrete support tickets. The tickets bear the symbolic function and can be flexibly organised to present problems in a variety of their different aspects by cutting-suturing and re-assembling the tickets in a variety of configurations. In this situation, abductive modelling of the production cycle appears to be the most suitable approach because the processes of creation and organisation of knowledge, as well as of problem composition are highly volatile and tend to evolve in unpredictable ways, making the upfront knowledge or testing impossible.[32] Therefore, both diffraction and abduction are included in the compositional methodology: the former as the view of the queer performativity of the software production labour,[33] and the latter as the way of creating operative models for production in situations of high uncertainty.[34]

Chapter 3 discusses how the compositional methodology notion of epistemic infrastructure is used in the topological treatment of operations in DevOps. This is discussed in three sections. The first section addresses the deployment pipeline as the topological machine. It understands the automatic production of space as a maximally auditable process and explains it through a computationally-informed delivery of the software components. The second section discusses the team topology principle, focusing on the benefits offered by the stream-aligned organisational pattern over the more traditional division into functional team types. Section three deals with the specifics of the circulation of knowledge in the software production system. It discusses in more detail such dispositions as the structural coincidence of organisations and the code of the software systems they produce, the tendency of software product to become a process, and the propensity of systems to disintegrate.

Chapter 4 zooms in on the specificity of negotiation of meaning in the problem space of production, which it finds to be intrinsically linked to the practice of audit. The chapter opens with an interlude that explains the empirical part of my research where I first encountered this challenge, and then turns to the discussion of the main facets of establishing the conditions for auditability. The first section defines collectivity in relation to the software production system, including the negotiations that occur in the intersection of the communities of practice and the organisation domain. In this context, the chapter briefly addresses the notion of affect as the deciding factor of much of the organisation's communication, which here needs to be addressed in terms of the risks it presents to the integration of knowledge, and more specifically, to audit. The second section aims to unpack the disruption orientation of the complex workflows through the continuity of dysfunction which, it argues, is a characterist-

---

[32] Peirce, 1955: 151.

[33] Barad, 2011.

[34] Magnani, 2009.

ic trait of most software systems. The dysfunction means the inability to fully repair the software system, and the necessity of treating its multivariate breakages as cases described in support tickets. The third section looks at the support ticket itself as a focal point of the disruption-oriented workflow that makes it possible to bring together the various criteria of production and enable the problem negotiation.

The ticket, which is a complex tool used in a variety of production contexts, appears here in only one of its aspects pertinent to the discussion of planning and review, as the audit tool. The practice of audit is discussed as distributed in its nature, due to the force of complexity that prevents it from developing a hierarchy and evolving around a central axis. The audit instead focuses on the control of control, that is, operating not on the data itself, which, due to the complexity of the domain, cannot be audited directly. Instead, audit requires a specific environment prepared for it – in the case of present research, a software production system – which means that it is capable of verifying only what such an abstracted view permits it to verify. This shortcoming of the systematic approach is taken into further consideration in the thesis's closing chapter.

The aim of Chapter 5 is to unpack the consequences of the complexity of the production system that are faced by its governance. I suggest that, like audit, governance also tends to become distributed under the pressure of the extreme complexity of the production context that it needs to regulate. The complexity effects are aggravated by what is referred to here as the falling cost of computation. The depreciation of hardware, which implies that businesses incur lesser costs for their compute, is evaluated here not only in terms of the opportunities, which is what professional DevOps sources typically focus on, but also the risks that organisations face, due to their inability to adapt to the continuous change and exponential surges in complexity of their software systems.

Summary

Summing up the introduction, the argument of this PhD thesis can be outlined as follows. Software capitalism creates profits by keeping the software in a state of perpetual disrepair. The valorisation of disrepair works by increasing the complexity of software products through either deferring the integration of tacit knowledge or employing explicit knowledge to create new abstraction layers. The organisation, in its effort to maintain the complexity equivalent to the software system, risks becoming unable to control and audit the production process. Evolving in the intersection of the market and organisation, production acts as an interface for resolving the contradictions between the two domains by enabling the circulation of knowledge between its two parts, the problem space of production and its epistemic infrastructure. The two main circulation moves here are deployments and integrations. Since

the deployments largely operate through freely scalable symbolic manipulations, and the integrations deal with embodied and affective tacit knowledge practices which are not as easy to scale, the organisation's governance mechanisms evolve to become distributed and adaptive. Starting from a general concern that software production teams usually suffer from unnecessary complications that the organisational protocols introduce to their work, the research question grows to become entangled with epistemological and methodological aspects of production. Towards the end of my PhD research, I sharpen my question to ask, what kind of approach the production needs to develop towards its epistemic infrastructure to plan and control in the uncertain and disruptive situation created by the software complexity?

Chapter 1. The software production system

The chapter's goal is to set up the overall context of this thesis as thinking about production in terms of systems theory. Such an outlook seems important because the disciplines that the present study works with, from management to organisational to cultural studies, predominantly engage with production in terms of systems. More importantly, it is necessary to introduce the systems optic in the beginning to grasp some of the notions that I use as building blocks in the later chapters: traction, momentum, reverse salient and conceptual integrity. Focussing on the systems of centralised type, Chapter 1 explores their benefits and shortcomings to prepare for the discussion of distributed systems in later chapters. I split the discussion into three sections. I begin by arguing that whether organised centrally or in a distributed way, the balanced view of the system needs to account for the system's traction, for the present study defined as the relation of change rate to the ability to control. Then I turn to the concerns associated with the loss of traction and the notion of software crisis that it brings and ask what kind of solutions the systems approach had offered to the crisis in the past. Here the discussion evolves around the two early computer scientists that contributed to organisational thinking in software production, Frederick Brooks and Melvin Conway. It explains the popularity of Agile methodology, now present as the industry standard in most IT organisations, by its flexibility, which allows it to deal with software complexity and maintain traction.

The chapter concludes by addressing the broad socio-economic context of current software production as *software capitalism*, which I define as a culturally-informed capitalist formation where shared knowledge is the primary source of value, and every firm is seen principally as a technology firm. The discussion of software capitalism looks at the three of its principles relevant to the present research. First, software can be construed simultaneously as a service and as a product, and involves labour which is both value-generating and serves the interests of the culture of software production. Second, such culture lies outside of the capitalist relation of value exchange, yet cannot be entirely disregarded from the study of such relations because it plays an important role in maintaining the system's traction. Third, the software capitalist production model tends to valorise complexity – in other words, it welcomes the disrepair because the disrepair creates interruptions which, in turn, allow the creation of new cases and expand the scope of production, which concomitantly expands the capital circulation with the potential to increase the financial gains.

Traction in technological systems

The interpretation of production in terms of systems is the usual way of developing or supporting software throughout the IT industry. It is necessarily adopted in the present study because the operations and production sources under investigation rely on such a systems view in all their activities within the production process. The parameters at stake here are the rate of change in a software production system's traction as the ability to control it. Traction sees the centralised system as a matter of active construction with the goal of creating a unified and coherent whole available for command and control, and the distributed system as a unity subordinate to the distributed governance and self-organisation. Management theorists Arash Azadegan and Kevin Dooley summarise the difference between the two types of systems through their approach to innovation and agent motivations. The centralised approach focuses on the resources and infrastructure relevant to a particular problem-solving event in proximity to the controlling mechanism. What is collected within the purview of control is seen as a valid contribution to the problem. In distributed systems, the agents, on the contrary, have more freedom to choose their location within the problem space and therefore can specialise and diversify in an ad hoc manner in relation to the problem space modifications.[35] In the distributed model, no priority is given to adherence to the core strategy, which empowers the self-organised units to discover new creative solutions.

In terms of motivation, centralised systems stimulate the agents by setting clear goals, requirements and responsibilities. In distributed systems, on the contrary, enthusiasm comes from the alignment of tasks to skills and reputation. The distributed approach is usually seen as a better fit for the spheres that are difficult to position around the centralised control. This is relevant to supply chain management and, pertinently to the present research, to complex production situations with constantly changing contexts, such as in software development. Considering the two types of systems in their application to production, the task becomes to understand the different treatments that software complexity gets in the two positions. This section examines the centralised approach in terms of the proposal that the traction in centralised administration tends to reduce as the systems grow progressively complex and become harder to change and control. Later chapters address other meanings of traction, such as the property of effective differentiation and integration of problems and assuring the auditability of production in the context of ongoing change.

---

[35] Azadegan and Dooley in Allen et al., 2011: 427–430.

Change process triggers and responses

The systematic approach to technics, equally common in the history and the sociology of technology as well as in professional operations sources, makes it possible to carry out the analysis of harmonious technological wholes comprised of specialised separate artefacts and people, complex yet open to required planning, quality control and other practices of audit. The analysis lends itself to the rationalisation of technical and organisational complexity that permeates all aspects of a social contract that the engineers, scientists and operations staff enter into for the duration of technical work. One example of such analysis is utilised by the historian Thomas P. Hughes, who in the 1980s had pioneered the application of systems theory to study technology. A technological system in this approach is perceived as a goal-oriented set of physical artefacts and concepts which shapes society in the same measure as it is shaped by it. The challenge is to consider technical and social events in their mutual evolution without leaning towards either a technologically- or a socially-deterministic position. The complexity of the system appears in this model linked to a system's functioning, with a caveat that it is not the sheer number of functions performed that contributes to the system's complexity, but rather the system's heterogeneity, which then leads to the definition of a complex system as a system where connections do not form an identifiable pattern.[36]

The two neighbouring terms, *technical* and *technological*, refer to the system's scope. *Technical* refers to phenomena and processes solely concerned with physical artefacts and software. *Technological* is a broader term that refers to a whole comprised of organisational forms and technical functions.[37] A technical system denotes the system's operation in the sense of its practical expression, while a technological one specifically addresses the associations between purely technical aspects of the system and the organisational entities linked to them. Organisational entities may include manufacturing firms, utility companies, financial bodies and other institutions taken in terms of the technological knowledge they have. The knowledge comes in different forms, including records and protocols, such as technical documentation or legislation, or various communication patterns, such as teaching programmes.[38] The associations between technical and organisational parts are dynamic in terms of their continuous movement from one act of problem-solving to another in the presence of human actors who invent, design and develop in a repeated sequence. In this relation, the production design lifecycle can be explained with DevOps as a pattern of mutually inflicted change that happens in the IT company and the software product through the circulation of knowledge between deployments and integrations. The

---

[36] Hughes, 2000: 454 citing Moses.

[37] Hughes in Bijker, Hughes and Pinch, 2012: 74–75.

[38] Ibid.: 45.

technological system acts as the mechanism for creating and resolving the problems that emerge in the process of applications of methods, however, inevitably adding complexity with every iteration. Innovation tends to follow the rotary movement of the technical and the organisational parts following one another: product management is inclined to select the technical components that best support the existing system structure or the organisational form, and the organisation inhabits its technology through communications.

It should be noted that not all change in systems, organisations or software systems end up with complexity increases. For example, in terms of strategy, decommissioning the system's components or features may result in lessening the complexity due to the exclusion of these parts from the production workflow. In the sphere of productive activities, some work intentionally aims at reducing complexity, such as *refactoring* – the process of optimising the existing body of code by altering its internal structure without changing its external behaviour. The present study, however, is interested in the reasons for complexity increase, and largely is not focused on the changes that simplify things, following up on the concern of Hughes that in real-world production most changes tend to add to complexity rather than reduce it.[39] The tendency for increases in technological systems with every change, in terms of components, as well as staff employed, is fundamental to the complexity problematics and is something that this thesis encounters in different facets which are dealt with as they arrive.

In this sense, looking at Hughes is informative, as it allows for tracing the trends which software systems have in common with the overall category of technological systems. One of these trends is a propensity to overcome the growth pains by tracking the increases in the system's load via the load factor, which is frequently followed by diversifying the resources used to reduce the utilisation expenses – an economic mix.[40] The load factor measures the ratio of average output to the maximum output, which makes it possible to adjust the supply during peak demands, for example, the use of electricity at different times of day, avoiding system overload. In cloud architecture, this factor is also used and is known as load balancing. In a similar vein, it describes how the available virtual machine instances are managed to reduce the amount of time required to serve the user requests, such as in the peak times of software use. Chapter 5 will return to the DevOps connotation of load balancing in its discussion of cognitive load. In the second instance, the economic mix utilises the load factor readings for cost optimisation. The steady base load is usually carried by the most economical means, and the peaks activate the less efficient ones, or, put differently, the costs are higher for the times when the service is in

---

[39] Hughes in Bijker, Hughes and Pinch, 2012: 50.

[40] Ibid.: 65.

most demand. While the load factor makes it possible to track the usage dynamics, the economic mix acts as the driver for change: when the system experiences higher demand, it undergoes more stress, and the processes are directed at expanding the system.[41] The reverse dynamic is also possible – when the demand is not enough, it becomes economically unviable to support the exceedingly large bandwidth, and thus the operations look to decrease the capacity. How effective operations can be in their modification attempts is addressed by another system's attribute, which is important for the present case: the system's momentum.

Momentum and balancing regression

While large systems are often more difficult to modify, the inertia, or *momentum,* is not a sign of the system's scale, but rather its maturity. As it is understood in the present study, momentum is a factor similar to a later notion of team *velocity* used in Agile. Where velocity measures the rate of completion within a particular situation of production, momentum is the rate of change within the system as a whole. Slowness in response to change does not always present a problem – in fact, a system can work extremely rapidly precisely due to its great momentum, supported by the organisational production culture, highly specialised teams or a workflow which is tightly adjusted to the production goals. Yet, since centrally-organised systems largely maintain control through approximating complexity with standards or integration of operations into the system, the overall structure may appear fragile when the problem criteria change or do not follow regular patterns. As Azadegan and Dooley observe, 'standardisation and integration can lead to reduced flexibility… [which] may lead to an inability to effectively respond to changes in the environment.'[42]

Throughout my fieldwork as a product lead, I have encountered that due to the effects of momentum, some of the organisation's top-down directives, even where they concerned minor issues, could not have been easy to carry out immediately. As an example, in one of the weekly departmental meetings, the stakeholders had voiced a concern that the last week's comments were still not addressed. From the executives' vantage point, the matter was carrying out a task pure and simple – which in this specific case was a retrieval of a quote for a new service that third-party contractors were planning to provide. However, the fact of the inability to do this task during the week made me reflect more broadly on how the production flow could be organised for prompt response to changes that required immediate action. After much consideration, I came to believe that the core problem was the missing technology executive role in the organisation's hierarchy that prevented the production department from reporting

---

[41] Hughes in Bijker, Hughes and Pinch, 2012: 66.

[42] Azadegan and Dooley in Allen et al., 2011: 421.

its activity to executive staff in a significant way. Indeed, at that time, the organisation did not have an executive level of technical expertise that would be sufficient to decide on the strategic delivery of the website's availability infrastructure. This, in turn, created a situation where the missing decisions on technology-based issues had to be filled ad hoc by the production team. Since effective decision-making was not entirely possible, due to the missing communication links, the production team struggled to resist the force of momentum, which dictated that the work carried out each week was scoped and scheduled for execution weeks in advance.

In Hughes' terms, for the system to acquire momentum, the technical and social factors have to be balanced: the high adherence of the system's social construction to its technological functioning creates high momentum and conversely, where either of the two factors pulls away, the momentum reduces. Some other factors that contribute to momentum are the sunk costs and assets, as well as the extent to which stakeholders of different forms, such as scientific or engineering communities, are invested in the system. An important outcome of my fieldwork has been that production team members tend to treat the benefits of momentum, without explicitly referring to it, as their basic production tool, which was mainly manifested in a more intensive style of communication at the initial phase of the project. For example, when the newly on-boarded team included players who did not know one another and were not familiar with the system, their first instinct was to have frequent and detailed discussions about the various technical aspects of the system among themselves. After the initial period of high bandwidth communication, I was registering a simultaneous increase in production momentum, and a shift in communication to a more casual reporting on the already established ways of dealing with new issues. Thinking about this self-organising tendency made me consider the difficulty of controlling it, which is addressed further in Chapter 5, and whether a notion of a system's traction, as the ratio of the system's complexity to the ability of its administration to control it, could be related to the system's momentum.

Hughes makes a distinction between innovation and invention as the primary causes of systemic change, treated differently by the bureaucracies, standards and flows of funding. Such institutional formations, he notes, create pressure for the system to reject the radical changes that do not conform to the existing operations, by perceiving the new features as technically crude or economically insecure.[43] Due to the numerous vested interests, large organisations are more likely to adopt the less risky path of *innovation*, where innovation is the process of change that upholds the existing status quo and plays along with the momentum of the system. On the contrary, changes of the *inventive* kind that demand disruptions to the ways of present protocols are generally met with resistance and are construed as det-

---

[43] Hughes in Bijker, Hughes and Pinch, 2012: 53.

rimental. The invention is a radical change, and, unlike innovation that seeks to find its place within the existing system, either requires a new system to be created or gets lost, falling out as the mature system moves on carried by its inertia. Innovation contributes to the system's overall tendency of growth and can be an outcome of activities that formalise the relations between the technical system and the domains of the organisation and the market. Where invention appears unexpected and comes with more risk of creating reverse salients, innovation enables planning since it can be recognised from the inside of the internal corporate structure. The ability to foresee the changes that innovation is going to bring also works to strengthen an organisational culture through reinforcing situated knowledge. In a centralised production system model, the risk of not being able to control increases together with the expansion of the problem space of production, and the efforts to compensate for the lack of control may lead to complexity surges that echo throughout the system. This happens due to suboptimal administrative strategies, such as overpopulated design efforts, discussed later in this chapter.

While the organisation and the technological system evolve mutually, where technical momentum is larger, the system shapes the organisation more than being shaped by it. The continuities established at the earlier stages of the system's development become reinforced, creating a shift in the cultural fabric of the organisation that brings a tendency towards technological determinism, or prioritisation of technological requirements over the organisational strategy, which may welcome innovation that supports the existing continuities, simultaneously precluding the system from any radical change that does not contribute to systems' direction of movement or challenge the status quo.

The notion of traction helps to understand how the complexity spikes increase the problem space and thereby stifle the organisational control, by bringing together the aspects of complexity and momentum pertinent to administrative control. The loss of traction occurs in the moments when the technological base has already changed but is still entangled with an old model of administration (Fig. 1). The circle in the left part of the diagram signifies the position of the firm's administrative approach relative to the market distribution techniques, represented by the circle on the right, with the distance between the two marking the boundaries of the problem space. The traction is at its best when the two are located on the horizontal line and are in full balance. Whenever a new approach is introduced in either one, the traction is negatively affected, the problem space expands and the complexity increases. This chapter predominantly focuses on traction in the context of a system as a unified, centralised and coherent whole enforced by the system builders, which, however, happens at the expense of diversity and pluralism.[44] Chapter 5 returns to the traction diagram to think about it otherwise, from the stand-

[44] Hughes in Bijker, Hughes and Pinch, 2012: 46.

point of distributed control, which makes it possible to equalise the audit practice with the fluctuating factors of processing and transmission of knowledge in the market domain through implementing the global protocols of administration within the situated knowledges on the level of teams and individual agents.

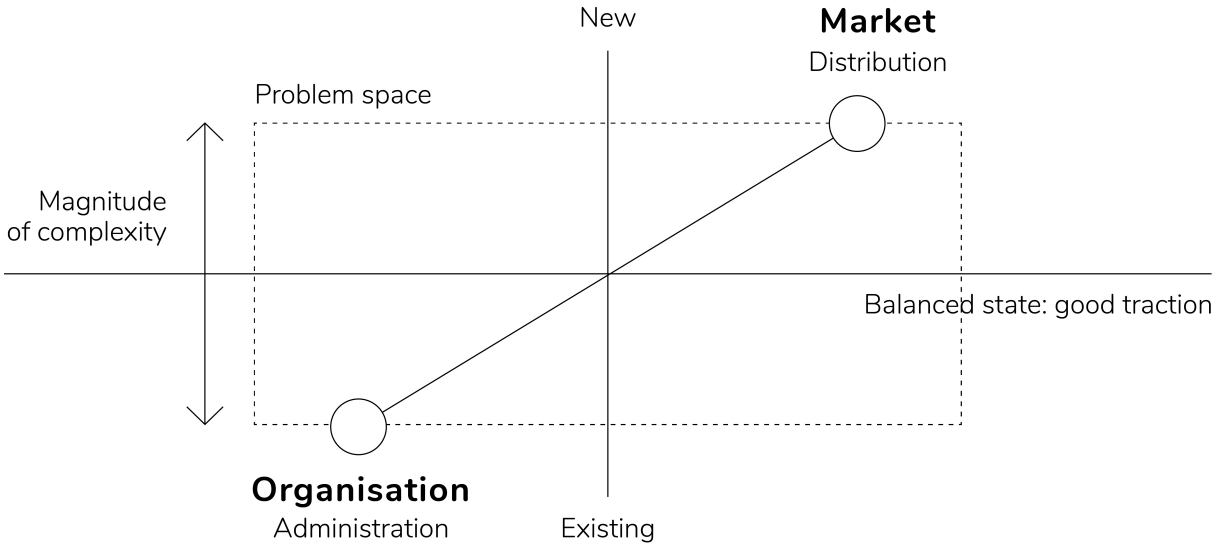

Fig. 1. Traction balance and the problem space.

Hughes refers to the topological ruptures caused in centralised systems by invention and innovation as *reverse salients* – a term otherwise used to refer to effects similar to a protrusion in a geometric figure. In system analysis, the term refers to the components that have fallen behind or are out of phase with the others, and thus appear as limiting the potential, causing frictions or otherwise hindering systemic efficiency. In a technical system of physical artefacts, engineers may change a characteristic of a power generator to improve its performance. This, in turn, requires a change in the motor that will utilise the improved power supply – meaning that the motor becomes a reverse salient at the moment the power generator is changed, and remains reverse salient until its characteristics – resistance, voltage, or amperage – altered according to the new system environment.[45] In other words, Hughes observes that innovation in fact, is capable of preventing the system in its entirety from achieving its development goals, whenever the development of any of its parts is insufficient. Furthermore, reverse salients are likely to reduce traction, because some of previously relevant production knowledge now has to be replaced, both technical and administrative, to warrant backward compatibility of new components, along with their audit.

---

[45] Hughes in Bijker, Hughes and Pinch, 2012: 67.

The roots of momentum are never uniquely technical or made explicit in the organisation's policies. The problem space of production evolves alongside its epistemic infrastructure or the abstract schema of what is possible to know about the problems. Yet, because problems always come as a result of stakeholder negotiations, the emergent knowledge reflects the variety of meanings present in social groups, tangential to the technological system. The negotiation proceeds through *diffracting* – cutting together-apart, to use the term of Karen Barad, explained later in this chapter – a variety of interests, resulting in something which may not fully benefit all parties. In the present case, this means that the complexity spikes in the moments when the stakeholders on either the organisation or the market side find it in their best interest to go to the contrary of what could seem like an appropriate technological solution. Hughes observes that the likelihood of such a situation increases in mature technological systems, where *the need for an organisation may often be a reverse salient*.[46]

What follows from this is that momentum is a quality not of the teams themselves, but of the production situations where the systems of high complexity meet the rigid administrative structures. For example, in my empirical work, the momentum tended to come from whichever part of the system had higher complexity: while the team itself was small and could negotiate without much difficulty, it would still have a lot of momentum, since the organisation's publishing platform it was dealing with was already a complex software product. Moreover, momentum spikes, whenever software complexity is added to compensate for the lack of flexibility in the administrative form, or, in other words, the technologists, create unnecessarily complex technological solutions to still be able to deliver a functioning system that creates great profits, albeit at greater costs. A production practice that ensures that previously implemented functionality of the system continues to work after introducing new features, known as *regression testing*, due to its high cost, can technically buttress the momentum. Regression can also be explained in terms of managing dependencies and can stifle the change because of the high amount of concomitant adjustments that will have to be made to neighbouring components after the proposed change has taken place. For example, if the Continuous Integration workflow is already realised through the use of GitHub in combination with third-party applications and a new Continuous Integration method becomes available, such as by using GitHub alone, the organisation may resist implementing the new method, even though the new method is easier to use. This will happen whenever the costs of making a change in the workflow and the associated regression testing are disproportionally risky and costly for the organisation's operations, which can be the case for large and small teams alike.

---

[46] Hughes in Bijker, Hughes and Pinch, 2012: 67.

On the organisational side, reverse salients could be identified through some of its *antipatterns*, – tendencies which serve to the detriment of the system's functioning. The two pertinent antipatterns – the team overpopulation and delegation at the expense of communication – are emphasised by the computer scientist Melvin Conway.[47] This process mirrors the regression in interpersonal communications, manifest in a 'this does not work for our real systems' kind of reasoning that business owners are inclined to put forward when confronted with production optimisation proposals. Such an interpretation of a system's momentum deals with sunk assets and sunk costs, understood in terms of organisation and cognition theorist Herbert Simon, according to which such entities equally appear in the form of the organisation's communications as well as brick-and-mortar assets, exerting a pressure of previous decisions that narrows down the options for future choices.[48] Chapter 5 will return to this aspect of momentum and its impact on system traction in terms of implied, or tacit, knowledge that it is linked to.

My empirical research has provided evidence that a strong link between the business value stream and deployment pipeline improves traction by making technological change easier to implement. Seeing the product both in terms of its business value and its entanglement with the technology has allowed me to introduce the changes despite the force of momentum pulling back into the existing status quo, specifically in the cases where I acted as a liaison between the organisation and the third-party IT service providers. My role at improving traction here meant that the work done by external contractors was not viewed as a ready-made product that is delivered when it is complete, but rather as a service that proceeds through the processes of regular communication and incremental steps of realisation so that when the new component or feature is complete, they are fully integrated into the body of the existing system. Therefore, I had to change the approach to my work with contractors to become a more explicit ambassador to represent the interests of the business. In this capacity, and having changed my job title to product lead, I was able to assess production situations in terms of the relation between the business value stream and the technological idiosyncrasies of the software system. This allowed me to find possibilities to promote technological change, for example, the realisation of a new search interface or migration to a new infrastructure, in the context where the habit of doing things in a certain way – the momentum forces – would otherwise disallow introduction of any new features for the sake of maintaining the existing status quo.

---

47 Conway, 1968: 31.

48 Simon, 1997: 148.

The crisis of control and the software crisis

In addition to change events and the attributes of velocity and momentum as discussed above, the notion of control is also crucial for understanding the system's traction dynamic. What is important for the notion of control, in Beniger's sense, as the activity of the goal-oriented persuasion, is that it is intrinsically linked through its history and etymology to the dual process of information processing and reciprocal communication between the controllers and the controlled.[49] In terms of information processing, the controller deals with comparing new information inputs to the existing database records, or, in DevOps, the policies that describe operations in their ideal shape. To do this, reciprocal communication is important, because it informs the control mechanism about the present condition of the system, and, reversely, transmits the directions the system needs to evolve according to those descriptions. Assuming the ideal feedback condition in which the channels of communication themselves relay the instructions perfectly, the effectiveness of control mechanisms in a production system will be defined by how well the receiving end is influenced. In other words, how fast operations and production teams respond to the demand for changes, and how well the forces of velocity and momentum are utilised to the ends of the predetermined goal.

The traction crises occur between the rate of change and the temporary dysfunctions, as illustrated in Fig. 1, due to the various deficits and excesses that happen as the technology works its way through the discontinuities between its components. The discontinuities either find their outcomes via the negotiations in the problem space or become crystallised in the system's epistemological infrastructure. Linking traction to the system discontinuities in this way helps to provide a common motif in the various historical meanings of the software crisis. It was first articulated at the NATO Software Engineering Conference in 1968 as the crisis of demand for new software, which outstripped the capacity of the combined engineering workforce to create it.[50] However, the demand for new code since that time has never reduced and has arguably become larger because later engineering had to deal not only with creating new code, but also with the support of the increasing body of existing code, and the compatibilities between the old and the new. Additionally, the notion of the 'code' itself is different in the component-based production deployments, which contain, besides the source code, the environments, configurations and databases. As the professionalisation of DevOps has brought together development and operations in the mid-2000s, it may have become possible to observe a somewhat reversed dynamic in the software crisis. Instead of demands for new software exceeding production capacity, the production capacity begins to act as a limit to what the computation has to offer. Looking at such changes

---

[49] Beniger, 1986: 8.

[50] Randell, 1996.

within the software crisis, one thing that may be observed as a constant is that the production and administration techniques always lag behind the rapidly evolving condition of technology, which continuously threatens to erode control. In this capacity, a software crisis is thus a kind of control crisis which exposes the audit techniques to situations of extreme uncertainty, which causes any events of planning and review strongly rely on all the possible means of codification and abstraction following the parsimony principle of agent knowledge acquisition, as Chapter 5 will explain further.

Such a control-based understanding of the software crisis makes it visible that centralised control is often not feasible, as it would involve too much overhead in audit activities. The overhead, to Azadegan and Dooley, is created by the centralised administration tendency to functional determination, or prioritising the knowledge and inventory that has been developed within the sphere of centralised control and to neglect what has been done outside of it.[51] This makes the system progressively hard to audit as the rate of change grows because the centrally adopted epistemological regime may not always be consistent with the infrastructural changes that occur in other parts of the system. In centrally organised projects, agents are expected to follow a predetermined sequence of project steps, such as knowledge acquisition, approval of requirements and execution. The motivation here lies in common goals and the clearing of milestones in the project's roadmap. Distributed control systems pay equal attention to a more diverse range of initiatives, due to the difference in motivation of its agents. Here the responsibilities are less deterministic in terms of project metrics and agents have more freedom, which promotes a more tight local alignment between the agent's skills and reputation and the tasks that the agent carries out.

In the centralised production process, it is therefore easier to invest in upfront planning, with consecutive rolling out of design and testing in a stage-gate way which makes it possible to audit the requirements, design and production of each system component, along with their integrations. As we have seen, however, such division into steps does not work well in all production situations. The distributed approach, instead of staged sequencing, relies on the bottom-up initiatives of agents who specialise and diversify the production events within a system, which allows access innovation across the whole spectrum of ideas that exist within the system and intensifies the circulation between the problem space and epistemology, ultimately improving the system's traction. Managing the software crisis in the real-world production context implies maintaining a balance between full centralisation and full distribution. This is required since, on the one hand, full centralisation risks stalling production due to the impossibility of managing all of its aspects. On the other hand, full fragmentation brings the risk of

---

[51] Azadegan and Dooley in Allen et al., 2011: 426.

the inability to follow any organisational agenda due to the excessive fluctuation of sub-processes carried out by agents, without recourse to an overall goal. Maintaining such balance is a matter of constant negotiation between the stakeholders in the context of change created by such factors as the falling cost of computation – matters that are discussed further in Chapters 4 and 5.

Incremental delivery in software production systems

As the previous section has shown, analysing technology in terms of centralised systems is fruitful for specific cases, such as mass manufacturing situations, because of the factors such as easier accountability, repeatability of contents and ability to plan. The question now arises, what kind of benefit might such a view offer to the study of software systems? Should IT production be analysed in terms of systems at all, or should it rather use some other criterion that would place it orthogonally to complexity effects? Starting from the foundational observations of Frederick Brooks that the essential complexity of software cannot be done away with, and comparing this to the treatment of complexity in Agile methodology, the section finds that a way of mitigating the complexity effects should not be sought within the system or outside of it. Instead, it is possible to grasp the software system and the organisation producing it as a unity, so that it is possible to understand the mechanisms that iteratively codify and abstract complexity to circumvent it within such a unity. Furthermore, while the systematic approach is well suited for strategic thinking when it comes to tactics, it may be favourable or even necessary to split the software and the organisation along the lines of communication, as discussed by Melvin Conway, the computer scientist active at the same period as Brooks.

Since the early software production efforts carried over the strong assumptions of mass industrial manufacturing, their failures can be attributed to two key features. On the one hand, the fact that the software system tends to reflect the problems of the organisation it interfaces with or Conway's Law. On the other hand, the organisational momentum at play in IT organisations does not allow them to adapt to rapid changes in technology. Brooks puts these two factors together when he claims that there can be no silver bullet in software production, that is, no one measure that would improve 'productivity, reliability and simplicity'.[52] According to the first factor, any complexity, even the one essential to the system's functioning, always signals the inconsistency between the software requirements, which are constantly changing, and the organisation's communications. Such inconsistency is a dysfunction that makes necessary the next round of production efforts in the system's oscillation between the disrepair and the integration of the system in its minimally deployable and auditable state. According to the second factor, however, this lag is unavoidable and essential, since the organisations are entangled in

---

[52] Brooks, 1995: 181.

their communications and have a certain momentum that does not allow them to change their communications as quickly as to adhere to the requirements. Therefore, the software system has to be adjusted to the organisation's structure, resulting in a more complex software system on the one hand, and hiring more staff on the other.

Essential and accidental complexity in organisation design

Since it is the function – and not simplicity – which serves as a measure of good design in software systems, the ultimate goal is not the simplicity alone at the expense of function, and therefore technical complexity is not a problem all by itself. It is, rather, an essential property of the software system.[53] The systems' function requires the heterogeneity of composition, and good design is a matter of finding a ratio between the ease of use and specifications met. A software system necessarily compounds diverse components, which may not have anything in common, yet can expand indefinitely and go into minute levels of detail due to their abstract nature – such property affords no limits to how complex the system can be, which is radically different from industrial production, where complexity is limited by the material nature of resources. In a situation where some part of complexity cannot be eliminated, yet the control has to be maintained, reducing the parts of complexity that are caused by non-functional factors, such as performance issues or suboptimal infrastructure, is a high priority. Frederick Brooks was the first to theorise the essential and accidental kinds of complexity. In the accidental complexity, Brooks distinguished the accidents of *conformity, changeability* and *invisibility*.

Software *conformity* and *changeability* are beyond the scope of the present research and characterise a software system's involvement with the stakeholders. The changes can be imprinted onto the system by the institutions and by economic, political and other systems it interacts with.[54] Beyond those interactions, the software system faces the requirements that come from its cultural involvement by way of its applications, relations between its users, and the limitations imposed on it by its hardware. Complexities of this kind are largely accidental because, as Chapter 4 discusses further, it requires great discipline, reinforced by the organisation's policies, not to add features that do not serve the primary design idea of the system. The *invisibility* kind of software complexity falls within the purview of the present argument and is related to the abductive modelling discussed in Chapter 2. In Brooks's terms, the software resists spatial representation because the relations between its components are too varied: 'software includes many diagrams at once.'[55] The difficulty of visual modelling of a software system, how-

---

[53] Brooks, 1995: 43.

[54] Ibid.: 184.

[55] Ibid.: 185.

ever, is a difficulty that can be addressed by clarifying what purpose the representation serves in the first place. When the goal is to grasp specific aspects of the system in their particular relationships, such a goal, as I find in Chapter 4, can be achieved using manipulative abduction. Such an abductive method works by engaging intuition to bridge the knowledge gaps in a situation of uncertainty.

Having observed that *essential* complexity cannot be done away with, Brooks likewise expresses little enthusiasm for the attempts at reducing *accidental* complexity such as time-sharing, object-oriented programming and artificial intelligence.[56] To him, any such attempts would only be able to reduce the impact of *accidental* factors, with *essential* difficulties remaining unchanged. Any real attack on the conceptual essence can only be carried out via such measures as further professionalisation of the domain, strong intentions in what to build, or incremental development in the process of actual production. As the present study later finds out, reducing accidents can bring enormous improvements, to the extent that operations methodologies such as Continuous Delivery can be seen as a proverbial silver bullet or a unified technique that organises the production process in a way that does not confront all of the complexity at the same time. Yet, it should be kept in mind a lot of change has happened since 1975, when Brooks first published his volume, most importantly, the Agile methodology and DevOps movement. Therefore, I retain Brooks's concepts of invisibility accidents and conceptual integrity, discussed in the next section, as the signposts to serve as comparison points between the centralised and distributed governance paradigms.

In terms of the process of production as the circulation of knowledge about the system from deployment to integration and back to deployment, a software crisis appears as a roadblock and a driver of organisational and technological change. Therefore, the terms on which such a crisis is negotiated are new in every case, based on the pertinent technologies and the component configurations. Historically, the first measure of improving such circulation was to create large quantities of code. As a result, this required new staff, which initially came in the shape of the formally trained software engineers, and later, less formally, via the grassroots movements as personal computers became more available. As more commercial and open-source code got created, the importance started to shift towards the task of integration. This, consequently, created a demand for production specialists who would deal with qual-

---

[56] This can be contrasted by a more optimistic view towards complexity expressed by the mathematician Vannevar Bush, US wartime Director of the Office of Scientific Research and Development, who wrote in his 1945 essay: 'Note the automatic telephone exchange, which has hundreds of thousands of contacts, and yet is reliable. A spider web of metal, sealed in a thin glass container, a wire heated to brilliant glow, in short, the thermionic tube of radio sets, is made by the hundred million, tossed about in packages, plugged into sockets – and it works!' (Bush, 1945: 102.)

ity control, cost estimations, verification and validation. The increases in operations staff, in turn, allowed for more deployment fluency, which came in the form of professionals dealing with data, configuration and environments. Such developments of the design lifecycle were a part of the general tendency of IT-based socio-economic formation, widely theorised in different sources and primarily referred to in this thesis as *software capitalism*, to improve circulation of value through gaining knowledge about itself, and more specifically by placing the business value stream close to the technology, up to the point where the business itself was no longer different from the software system it runs – e.g., the bank is merely an IT company with a banking license – the organisation becoming a sophisticated tool for capturing and processing knowledge.[57]

Structured and Agile production methods

Second, a journey towards effective decision-making on what to build could be illustrated via the two examples, Structured Design (SD) and Agile methodology, in terms of their treatment of *conceptual integrity*, a paradigm in software production design that prescribed the entirety of the work to be carried out by a single mind, supported by the organisation, who has a perfect detailed vision of the product as a consolidated entity that can be described by a set of requirements.

In the first place, SD, which became popular in the 1970s, in the account of the historian Martin Campbell-Kelly, had been one of the more successful attempts at a centralised approach to software production system governance.[58] It replicated the waterfall approach of the assembly line operation, prescribing all of the specifications to be written in advance, with design, testing and deployment stages following one another in an extremely rigorously structured way. The rationale for prohibiting the change requests, after the sign-off of the requirements, was to address the slowing down of the releases via *scope creep*, an antipattern which is manifest in the unplanned additions to the project scope after initialisation. The changes had to be taken seriously because the whole of the process, in line with the conceptual integrity principle, was held in the mind of the team leader, and thereby the scope was limited to how much new knowledge one mind could process. The leader was supported by a compact team of support staff, which made the model akin to a surgical operation. *The surgeon* at the top of the hierarchy is the one mind that brings the project together, with the assistance of *the co-pilot*. On the next level of the hierarchy, there are the administrator and the editor. The former is responsible for the operations, while the latter creates the main body of documentation and technical reports. One more level below are the *programme clerk*, who maintains the catalogue of all human and machine-readable

---

[57] Kim et al., 2016: 8.

[58] Campbell-Kelly et al., 2014: 185.

texts, the *toolsmith* who writes custom utilities, procedures and libraries, the *language lawyer* who edits the programme syntax, and finally the *tester*.[59]

Through such a presentation, it becomes possible to grasp a specific flavour of software crisis that the Structured Design paradigm was vulnerable to. To use Brooks's formulation, any large software production system designed in such a way is trapped in the allegorical *tar pit*, a problematic situation materialising out of the numerous diverse interrelations within the system which are hard to discern.[60] The risk of the tar pit is particularly prominent in large-scale production contexts, where the interrelations are too complex. Hiring more programmers does not help to reduce production time, and only seems to increase the number of communication links and the complexity of their interrelation. In cases of automobile production managed according to the algorithmically defined worker behaviours as prescribed in the Taylorist paradigm, a *man-month* metric is effective as a unit for measuring the size of a job. Such a *man-month* assumes that any problem can be broken down into several discrete units that could then be estimated separately, giving a reliable scale for measuring productivity. In the context of software production, however, as the team at IBM found out, man-month was a 'dangerous and deceptive myth,'[61] and adding engineers to the software project would not yield an immediate rise in productivity. This makes it possible to suggest that high-complexity production is different from factory-style manufacturing in that it does not scale in the same way and requires additional communication efforts, often in the form of developing a certain degree of distributed audit, as the thesis finds out in Chapter 4, to achieve the productivity increases.

The complexity of interactions within the collective encountered by the Structured Design efforts revealed a crucial difference between software production and the industrial model described in Marx: a greater number of engineers provide more useful outcomes in less time only when they carry out the work individually, with no communication between them. Needless to say, in collective production of complex artefacts the communication is by all means required, and in some cases, such as in training and coordination of efforts, cannot be split for concurrent execution, and therefore requires an amount of time that cannot be reduced. Furthermore, the interrelation between team members is another, and by far the most deceptive, metric that inevitably exceeds time estimations. The metric, also known as Brooks's Law, demonstrates that the time required to bring new staff up to speed increases as the staff numbers grow – 'adding staff to the project makes it later.'[62] This results in further delays that make it

---

[59] Brooks, 1995: 32–35.

[60] Ibid.: 4.

[61] Brooks, 1995: 16.

[62] Ibid.: 25.

necessary to bring even more people in. Despite a sea change that has taken place in production system design since the 1950s, the complexity of communication in software teams is still a great risk, which is met by a serious treatment of this matter by current DevOps. The DevOps theorist Matthew Skelton acknowledges that collaboration is usually perceived as expensive, something that needs to be restricted to well-specified cases and formalised as an explicit activity aimed at producing specific measurable value.[63]

Despite the benefits of speed and clarity of the single-mind approach, the sheer fact that the software is delivered to the requirements written in advance implies that the system is not as fast and flexible as the changing environment to which it is deployed. On the one hand, prohibiting the changes of specifications makes the integration more difficult, since should there occur any change in the systems or components that the requirements relied on, the finished artefact will have to be returned to the production cycle to account for such changes. On the other hand, the long design stage with infrequent deployments makes testing harder, more expensive and not extremely effective because such testing is done outside of the context of the error, and therefore requires additional research. Another risk is that if testing is a long and strenuous process, teams that work on tight deadlines cannot afford to test often.

The search for new design tactics that would allow avoiding such risks by incremental delivery in one way or another eventually led to abandoning the Structured Design approach to a new Agile methodology. The Agile movement was started in the early 2000s by a group of analysts and software developers with the aim of adapting the lightweight practices that were used at that time for small-scale projects, to heavyweight software production systems, and to replace Structured Design and other waterfall-based techniques. The three Agile ideas have to be mentioned in this summary. The first is something that Frederick Brooks formulated earlier as incremental development[64] and what the DevOps advocate David Farley later theorised as Continuous Delivery. In essence, working software is to be delivered frequently, in a cadence ranging from a couple of weeks to a couple of months, with a preference for the shorter timescale – in Farley's case, as the later chapter of this thesis shows, deliveries should be done as frequently as possible, to make the release a low-risk activity.[65] Second, as observed by Gene Kim, the team has to be small-scale and self-motivated and there needs to be a high-trust management model that is well adapted to the delivery in small batches.[66] The third and by far

---

63 Skelton and Pais, 2019: Ch.8.

64 Brooks, 1995: 200.

65 Humble and Farley, 2010: 280.

66 Kim et al., 2016: 428.

the most important feature of Agile is that it mainly deals with ethics of production, and is voiced by the production team, rather than by any other organisation's division.

Approaching production in the abstract is something that makes Agile applicable in a variety of organisational settings because it does not prescribe the production to be carried out in any specific way. Instead, the design of the production system is seen as largely instructed by the organisational structures it is embedded in. Simultaneously, any intentional efforts of implementing Agile in a form more rigid than described in the manifesto either leads to replicating routines without deriving any specific value or ends up being silently or explicitly abandoned for other, often suboptimal, tactics. One example of such attempts is Scrum, a governance-oriented Agile framework that puts an emphasis on metrics, such as assigning business value points to specific development tickets and estimates the overall value delivered by how many of the tickets were completed in a given period. Despite the popularity of Scrum, its real-life application demonstrates that it cannot be sustained without extreme management vigilance, which in the long run appears to merely produce glory metrics and not effectively designed software production systems. In his support of general Agile methodology, Farley argues that Scrum is not necessary for all Agile workflows and lists three main reasons why attempts to adopt it usually fail. One is that the leadership's commitment to Scrum tends to fade as they realise that it persistently makes them face inconvenient truths about the system's issues. The other reason is that Scrum is often used as a replacement for good engineering, which means that organisations claim to use Scrum without adopting its constituent Agile practices, such as test-driven development, refactoring and Continuous Integration. Lastly, organisations tend to change Scrum to fit their particular context too early in the adoption phase. This means they don't have a chance to learn from Scrum in its original form, and slowly drift back to their usual way of doing things while continuing to claim they use Scrum.[67]

As an illustration of the field use of Scrum, I was involved with the team using it in my capacity as a product lead during a large platform migration project. This meant that I was the customer representative working in collaboration with the third-party suppliers who used Scrum as their production method. This approach had rigidly prescribed the tools, methods and cadence of production activities, however, after the initial part of work was completed, the rigidity came under pressure due to the unstructured flow of feedback that the suppliers were confronted with from the very first moment when such feedback was made possible. Some of the feedback, including the construction of the home page, had to be incorporated into the original scope, while some others, such as the construction of the web-

---

[67] Humble and Farley, 2010: 427.

site search function, had to be postponed until the later release.[68] Eventually, under the burden of the scope creep, the migration was evaluated as delivered only partially, and the new version was launched as an intermediary version. Post-launch, my task as a product lead was to create a production approach that would be a better fit for the organisation's structure and communication.

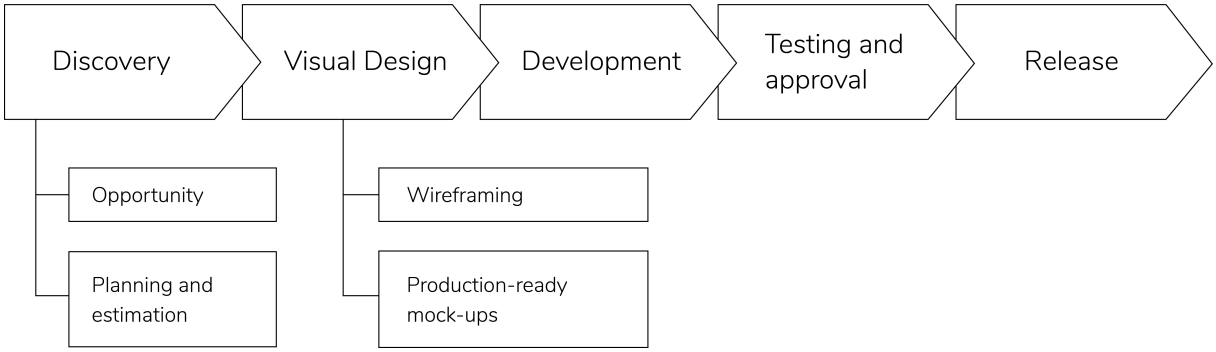

Fig. 2. The production pipeline.

Since each instance of software complexity arises out of communication challenges in the interactions between the stakeholders in the problem space of production, each system has its unique case of complexity which arises out of these interactions. This uniqueness explains the popularity of Agile over SD as the set of guiding principles flexible enough to allow production teams to circumvent some of the complexity challenges. Agile, however, is largely tactical, meaning that it helps teams to deliver rapidly and iterate in terms of software deployments and integration. Waterfall sequencing is useful, as the General Introduction proposes, in terms of strategic planning on a wider scale. For example, the project delivery pattern in the form that it has evolved in my fieldwork at JX, consisted of phases that followed one another in a set sequence, or a pipeline, which resembled a stage-gate approach (Fig. 2). Each project was initialised via a discovery phase, which included identifying the opportunity, initial planning and preliminary estimation of associated production costs. This was followed by the visual design stage, which started with creating an initial set of wireframes that allowed the content and marketing teams to clarify the feature requirements before spending resources on extensive user experience or graphic interface design. The stage concluded with the production-ready mock-ups, which were defined as the design specifications used by webmasters to verify that the work was done as requested. After the visual design, the feature went through development and testing stages and was eventually released. The release meant that all teams agreed on how the feature fits into the overall look and strategy of the system and that it was approved to be deployed to the production environment as an integral part of the system's source code.

---

[68] Some specific aspects of this work are described in Appendix, CS12 and CS16.

Importantly, the pipeline stages would remain the same for the production team regardless of the project content. To take an example of the online film festival, one of the larger JX projects, the discovery phase would yield a rough production plan based on the initial data collection, which in turn enabled the creation of a roadmap, or a project Gantt (Fig. 3). In this specific case, the Gantt shows the work split into three general parts that were to be delivered separately: festival open call, film festival itself and the awards, to be released after the festival is over, as the archival version of the project. The first phase would have the objective of collecting filmmaker submissions and assembling the festival jury, and would therefore focus on the contact form. The second stage required the integration of a film streaming platform, which would have ticket sales options and would allow the jury to rate the films. The final stage would include removing all the film-viewing functionality and creating a new landing page to celebrate the winners. While the three phases were unique to this project, each of the phases still had the same base stages, coming from the stage of discovery to creating wireframes, followed by design, development, testing and release as per the general production pipeline of Fig. 2.[69]

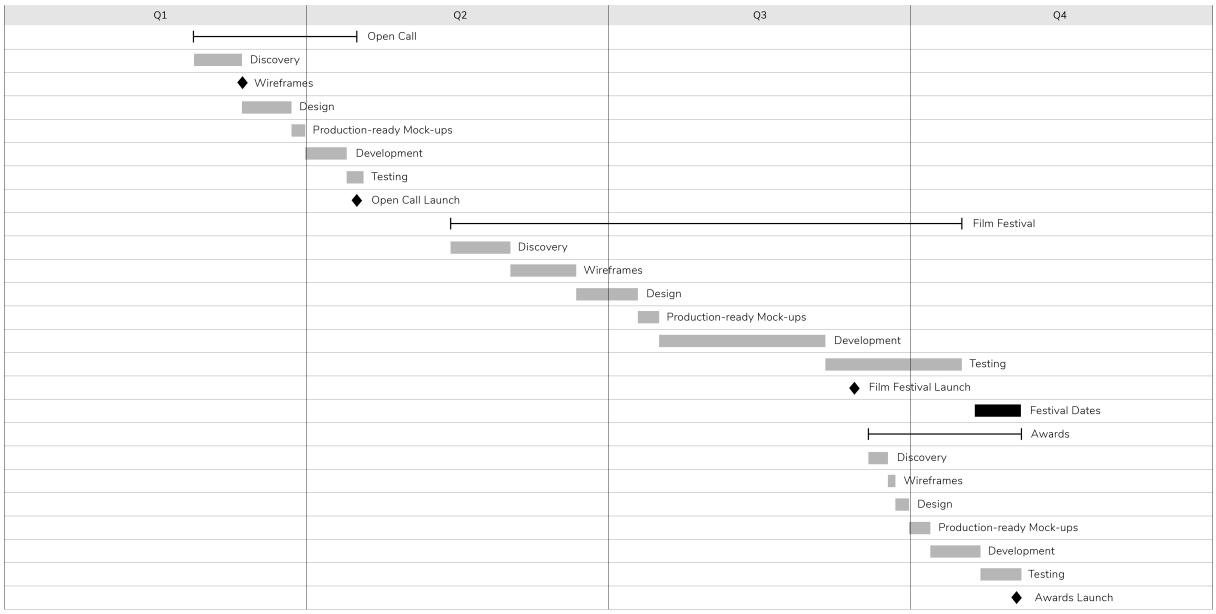

Fig. 3. Production phases and releases.

One formal benefit of incremental delivery is that some of the complexity can be avoided through the use of components with standardised interfaces which can be produced and deployed by separate teams either within or outside of the organisation. As soon as the system is split into components, it is technically possible to address the functionality of each building block via the cycles of deployment and integration. The real-world scenarios, however, are not purely technical and therefore not as clear-

[69] See Appendix, CS1.

cut as the product roadmaps. From my empirical study, I have learnt that sometimes the incremental approach can also be damaging, in the bigger picture of a project portfolio, if the delivery of parts is not timely. At one point in my practical work, the design and marketing departments had proposed updates to the information architecture. The updates were envisioned in the form of new sections of the website that would interactively cluster the information around geographic locations – something that later became known as City and Country pages. After more analysis, however, it was understood that the pages required additional work on other platform functions, such as the search algorithm, which was to be utilised in the task of sorting and grouping the database entries. Consequently, the project was broken into five parts, according to the rule of incremental delivery. Due to the involvement of staff in other more urgent projects, however, the five stages were spread over two following years, which meant that many of the original requirements were outdated. For example, the COVID-19 pandemic that broke out in 2020 meant that City and Country pages would not be a priority, as the travel sections of the JX website were not in as much demand as they were in pre-pandemic times.

Where the changes could be rolled out more slowly, however, the method of incremental delivery had proven to be more effective. For example, in the process of improving main navigation, other UX and particularly refactoring. Refactoring, by its definition, is the process of optimising the existing body of code by altering its internal structure without changing its external behaviour and can be delivered incrementally since usually no strict adherence to deadlines is required. Similarly, main navigation was designed to extend on the previous UX and thus was not something that required frequent changes and could be planned and released flexibly around other projects the digital production team was working on.

To summarise, a software crisis witnessed by Brooks could be attributed to the misbalance introduced by the limit of how much information a human mind can hold. In this light, the *conceptual integrity* that Brooks proposes as a remedy might seem unachievable in situations where the complexity of the code tends to expand indefinitely. In the later chapter, the present research views the conceptual integrity problem through the lens of the concept of *cognitive load*, as it is discussed in Matthew Skelton.[70] The concept, as adopted in some of the organisation studies scholarship from cognitive science, explains some of the current DevOps research trends that deal with the human mind's limits of acquiring new knowledge. For example, analysing the team's functioning in terms of the cognitive load of its members makes it possible to create a set of tangible metrics for assessing the overall team performance, and to create new strategies for distributing the load. Adopting the lens of cognitive scholar Edwin

---

[70] Cf. Skelton and Pais, 2019: Ch.1.

Hutchins, it becomes possible to evaluate the dependence of cognitive load and the character of cross-team communications, which is relevant for planning, review and other audit practices,[71] – something that Chapter 5 deals with in more detail.

Principles of software capitalism

Software capitalism (SC) is a culturally-informed capitalist formation with two characteristics pertinent to the study of production. One is that shared knowledge is the primary source of value. Granted that knowledge is inseparable from its subjects, that is, those who have the knowledge, the domain of value relations is entangled with organisational cultures. The other characteristic is that every firm is seen principally as a technology firm, as Kim notes, with the dual-core of the organisation present as the unity of the business value stream and the *technology value stream,*[72] in which context 'a bank is just an IT company with a banking license.'[73] I adopt such a two-fold construction in my research to understand it as a formative feature of the software capitalist production model. The primary function of such a model is the conversion of business hypothesis to customer value via a technology-enabled service, manifest in the deployment pipeline of the production system. In other words, for DevOps, every company is primarily perceived as a technology company, regardless of what line of business it may see itself in.

The historical roadmap for SC is adopted here from the organisational scholar Nigel Thrift's definition of *soft capitalism*, which fits the capitalist paradigm shift into the long period between the 1960s and early 1990s. The shift was made possible due to the economic and political factors. In terms of the economy, the period is signified by the decline and eventual collapse of the post-war Bretton Woods system that held gold as the basis for US dollar value and tethered the other currencies to the dollar itself. Politically, the period ended with the fall of the Berlin Wall. These events and the mind shifts they produced in the ideas about economy and politics across the world, Thrift argues, had in turn resulted in a tectonic shift within the global model of production and exchange. A previous more 'hard' version of capitalism, associated with the modernist top-down style of administration and scientific management, was confronted by the new 'soft' version that had a more flexible approach focused on planning and administration of human capital as the emotional and creative resource.[74] The change of understanding capital in terms of human characteristics is attributed to the growing importance of know-

---

71 Hutchins, 2000: 224.

72 Kim et al., 2016: 8.

73 Ibid.:xxvii citing Little.

74 Thrift, 2005: 31.

ledge in the creation of value, and, in turn, to the fact that the acquisition of knowledge escapes direct appropriation and instead has to be assimilated diffractively through performative labour practices. More specifically, in DevOps, one way of describing the novelty of such practices would be through the negotiations between the epistemology, the technology and the methodology of production.

In terms of epistemology, the capitalist model of mass manufacturing treats the problem space of production as the space where the ways of knowing are certain and well-balanced, while in SC, the same domain of practice or experience is understood in different and inconsistent ways.[75] The strategy is decentred and the popularity of a position is no longer assumed as a deciding factor of its validity, which means that the solutions are not prescribed in a top-down way, but are produced in a situated fashion through the ad hoc negotiations between agents. In terms of technology, there is a competitive advantage present in the new learning possibilities offered by the advancement of communication channels. In a situation where knowledge is linked to value, the competition focuses on research, teaching and learning, which sees organisations as learning units, with the scope of their learning including internal investigations about themselves. Some of the industry's professional occupations, such as business analytics, are uniquely engaged with research activities within the institutions. Learning makes it possible to facilitate knowledge acquisition and spend less time processing the knowledge that has been acquired. As such, it becomes a priority in the business sphere, which, as the three tendencies illustrate, creates a context, which largely characterises SC, where knowledge bears the principal business value. The strategic tendency of businesses towards improving their learning capacities binds them to the sphere of academic research. As listed above, the interdisciplinarity of methods makes them applicable in both academia and business.

Reversely, the extensive academic study of business is also taking place, which leads to new intensive industrial research, including the use of complexity theory in management studies, such as the assimilation of knowledge through agent-based modelling that Chapter 5 discusses in relation to the work of management scholar Max Boisot. The studies are necessary, not least because the problems faced by capitalism in the current shape of knowledge production are data-rich and thus tend to grow exponentially complex over time. Furthermore, the agent-based research agenda opens an avenue for the process-based, adaptive view of the system, since preventing change in epistemological infrastructures is both difficult and does little for capitalist valorisation. In the face of the falling cost of computation, the change required to acquire the user base and scale up production volume is no longer a risk. Instead, it becomes an opportunity because scaling the infrastructure in software is flexible and can be

---

[75] Thrift, 2005: 26.

adjusted according to demand, and falls uniquely into the purview of DevOps, with no significant change to the rest of the firm's operations. While any change in the production system introduces complexity, the trade-off of making a qualitative change to beat the competition is more lucrative to the business operating with virtual assets than it is in brick-and-mortar manufacturing, where quantitative change may often be easier. In other words, in software capitalism, delivering more of the same product concerns distribution rather than production.

Lastly, there are epistemological and technological shifts in methodology that the study of software capitalist mode of production needs to take into account. On the one hand, the new epistemologies demand new production methods dealing with the acquisition and integration of knowledge. On the other hand, the increasing capacities of communication channels create an influx of large quantities of knowledge materials in the form of components, features, services and other inventory, which are heterogenous, incomplete and highly perishable and therefore require new methods of dealing with it. The methods in this situation benefit from crossing the borders of their disciplines, with no methods prioritised over others, resulting in more engaged interdisciplinary encounters. Pertinently, DevOps as a professional occupation can be drawn here as an example of the successful adoption of development methods in the domain of operations. It is an interdisciplinary method of inquiry into the software capitalist mode of production because it carries out research into the organisations of technical and human knowledge[76] in the context of a firm in which the business value stream coincides with the deployment pipeline. Furthermore, as the next section explains, approaching DevOps in terms of queer performativity of its practice raises the questions of accountability and responsibility of knowledges negotiated in the problem space of production, leaving behind any previous assumptions and without giving priorities to any approaches.[77]

The market and the organisation, in their coming together in the problem space of production, thus spring from it in very different directions. The market interprets the logic of interaction via the value as the basis of the exchange. The organisation is concerned with creating relations within production itself and is thus shifting towards the production of its internal cultural values, which may not be directly related to the exchange value recognised by the market realm. Instead, as the organisation and cognition theorist Herbert Simon observes, the work of an administrator involves taking decisions, on the one hand, about the organisation's structure, and on the other hand, about the content of the work

---

[76] Barad, 2007: 93.

[77] Ibid.

carried out.[78] While in the software capitalist juncture, both of these matters fall into the duties of DevOps, the divide between the two appears less clear-cut. The former factor is blurred by the effects of Conway's Law, causing the structure of organisations to coincide with the structure of its software production system, to become stream-aligned, as Chapter 4 discusses further. This, to some extent, makes it impossible to think about the content separately from the structure. Instead, both matters co-evolve and adapt to one another, which is becoming possible as both structures become more loosely connected, ready to disperse and recombine according to the topological complexity demands. To grasp the involvement of DevOps that realises the production system in the software capitalism formation, the rest of the chapter focuses on the three themes. First, the stakeholder involvement in the problem space of production has to be explained as a service, or performative labour. Second, software capitalist production needs to be understood as cultural production. And lastly, it needs to be made explicit that the software production model derives its surplus value through valorising computation.

Performativity of software production labour

In this section, I approach the performative aspect of production through the following steps. I begin with some foundational thoughts of Marx through the lens of the Italian Autonomist thinker Paolo Virno, who contributed to the development of the concept of performative labour. I complement these ideas with the view of operations studies, where, according to the sociologists Sandro Mezzadra and Brett Neilson, the performative view might not be too relevant due to the macroscopic scale the operations are dealing on. In the next step, however, I find that the performativity of software as a service cannot be easily separated in DevOps from the software as a product of labour. Instead, due to the prominent presence of affect in the production of tacit knowledge, the labour in high complexity scenarios has to be viewed, in terms of the philosopher Karen Barad, in its queer performativity, which makes it possible to avoid the split between the representation and what it aims to represent. This, in turn, focuses my study instead on the performative as the relation prior to its terms, and instead something that constitutes them.[79] I find the empirical way of accessing this relation in the example of Jira, the support ticket software, and the practice of user acceptance testing (UAT). Both of these phenomena act as interfaces for coordinating the dependencies in the domain of organisation, the identification of market value, and the performative labour in production teams.

In its more general definition, the performative aspect of production can be sketched out through the understanding of performance as something where the event of work cannot be separated from its

---

[78] Simon, 1997: 327.

[79] Barad, 2003: 804.

product. In Marxian terms, performance appears as the waged, but unproductive labour, where the primary concern is the potential of deriving the surplus value from the activity in the absence of the autonomous end product.[80] To Virno, for performative labour to appear as the event of production, it needs to occur in a specific context that brings together the participants in the same space and time. In other words, performative production has to be publicly organised, which means it is manifest as political activity and should be accessed in terms of such structural characteristics as the lack of an end product and the necessity to be publicly exposed.[81]

In terms of its public organisation, the circuit of the software production system is only brought into motion once all of the stakeholder groups, including business owners, production staff and product users are brought together and begin to communicate within the same space, specifically, in the problem space of production. Such space is topologically defined in terms of continuities and borders, rather than in fixed metrics because the geography of hardware resources, the location of company premises and the whereabouts of staff can be varied. Once the stakeholders are brought together, they enter into a specific regime of problem negotiation which is regulated by the demands of audit, as further discussed in Chapter 4. The auditability dimension makes it important to integrate the knowledge back into the EIAC by making it explicit, which is often problematic due to the tendency of agents to develop a rich body of tacitly informed practices. Returning to the notion of performativity as the lens that recognise labour in its affective and tacit aspects, at this moment it is important to note that the software system can also be understood in its performative capacity, as something that operates through disrepair, and is always present as legacy code. As it is known to developers, any code becomes legacy code once it is written and deployed,[82] or, in other words, during deployment the code enters into a performative relation with other human and non-human parts of the system, making it possible to computationally interpret and transmit the relations of production contained in the infrastructure into the problem space for the next cycle of system development. Therefore, any software system needs to be considered as perished in the moment of its production, or, conversely, once the system is not maintained and any of the stakeholder groups are missing, the system, similarly to any other type of performative production, turns into a memory, or a fixed document of the product's condition at the moment of its last deployment.

---

[80] Marx, 1990: 1048.

[81] Virno, 2004: 54.

[82] Cf. Spolsky, 2000.

The study of operations approaches the performative aspect of production with criticism as something which is not entirely sufficient to address the wide problematics that the operations of capital deal with. Political theorists Sandro Mezzadra and Brett Neilson define an *operation* in this context as the frame that brings into focus the concatenation of social activities, technical codes and the mediating equipment which makes the operation possible.[83] Due to its macroscopic logic that acts to establish the patterns for the more nuanced production activities, the operation frame entangles the collectivities of production within the networks of its outsides, consisting of institutional entities, either within the same organisation or the wider organisational and market contexts. Contrary to this, the study of performativity, the argument goes, is self-referential in the sense that it is directed inwards into the problem space of production. It is political in its capacity to enter the negotiation of collective subjectivity through the conflictual relationships of its affective and embodied aspects, yet precisely due to its self-referential character provides little in terms of understanding the orientations of the subject in relation to the external relations of capitalist operations.[84] Performativity, therefore, is only capable of providing the potential triggers for specific processes that occur on the plane of operations.

The impossibility of the performative to activate on the macroscopic scale that Mezzadra and Neilson describe in some way resonates with the notion of momentum in large technical systems described by Thomas P. Hughes. The case for performative labour in high-complexity production, however, could be approached in terms of traction rather than momentum, which would shift the focus from operational maturity to governance. As mentioned earlier, the shift to distributed governance enables the systems to improve their resilience in the face of radical change, while being able to maintain auditability. This means that instead of attempting to implement the performative style of operations on the macro scale, the situation of high complexity causes the system to substitute it for the multiplicity of meso- and microscopic local performative events. While a more detailed account would be beneficial to describe the mechanisms of system mobility from the momentum to traction-based operative principle, this theme lies outside of my present scope and could provide a rich context for future research. At this point, it is necessary to address an underlying principle that creates such a possibility, which is the queer performativity of DevOps.

The queerness of operations in this context should be understood through the notion of *diffraction* of the philosopher Karen Barad. While Chapter 2 provides a more nuanced discussion of this view as a methodological underpinning of my project, the present discussion of the performativity of labour in

---

[83] Mezzadra and Neilson, 2019: 67.

[84] Ibid.: 248.

software production merits some preliminary remarks. Barad borrows the diffractive approach from physics, where it is used to study the distortion and interference of waves, and applies it in queer theory to shift the focus of performativity from specifically human affairs to the study of the practices of differentiating that non-human entities engage in the encounters with their environments. As Barad puts it, 'the point is not merely to include nonhumans as well as humans as actors or agents of change, but rather to find ways to think about the nature of causality, origin, relationality, and change.'[85] A notion of diffraction, therefore, can be interpreted as the study which occupies multiple vantage points simultaneously, yielding a rich view of the matters under investigation, specifically in terms of their non-fixed identity and across the assumed boundaries, such as the rigidity of distinction between human and non-human entities. In another place, Barad elucidates a principle of *cutting together-apart* as the essence of the diffractive method which entails identifying the differences which are crucial for the functioning of assemblages, within the junctures of their constituent parts. Cutting together-apart constitutes one move with the aim of making such differences clear: 'diffraction is not merely about differences, and certainly not differences in any absolute sense, but about the entangled nature of differences that matter … Diffraction is a material practice for making a difference, for topologically reconfiguring connections'.[86] The differences therefore are defined through the meanings they have in specific contexts, or, as the next chapter will further discuss, in terms of their topological orientation.

Queering DevOps, or in other words, approaching the high-complexity production event in terms of its simultaneously differential appearance as a process and a product affirms the software system's contradictory position concerning the value-control axes. It allows acknowledging that labour in software capitalism is fundamentally diffracted, in the sense that it is productive of business value to various degrees, depending on how the production lifecycle is viewed. Two examples may help to illustrate such queer performativity. In the first case, Atlassian Jira is a piece of production software, which has existence after the moment of its production in the form of a software release, which is presented to the users, who then engage with the product independently. Simultaneously, Jira appears as a service, in terms of the maintenance work and development of new features performed by the production team at Atlassian, which is inseparable from the process of software delivery.

In the second case, production is diffractively cut together-apart within the organisation at the moment of the user acceptance testing (UAT). In Agile, UAT is the final testing stage which requires a limited number of stakeholders, typically internal staff of the organisation, to use the product in real-life sce-

---

85 Barad, 2011: 124.

86 Barad, 2007: 381.

narios to test how the software performs and submit the feedback to the production team. To make UAT possible, the product has to appear to the stakeholders as a stable and independent entity which they can access and feed back on. Simultaneously, however, the testing activity itself does not constitute the delivery of the software with the purpose of monetary gain and has no meaning without the process of negotiation between the business owners and production staff, which is, in fact, its main purpose. Therefore, the diffractive view of UAT as a production event would interpret it as a compound activity which is essentially performative, yet impossible without its simultaneous presence as a product. Another important aspect of labour performativity is further elaborated in Chapter 4, when the discussion turns to the stream-aligned production team topology paradigm, often utilised in DevOps. The paradigm sees the teams as standalone entities which are continuously and flexibly assembled based on the complexity requirements of a particular production situation. X-as-a-service is a mode of team interaction where they consume or provide something with minimal interaction[87] through a separately managed interface or exchange protocol.

Summarising the queer performative specificity of labour in software which this section works with, it might be possible to suggest that the self-referentiality of performative relations could be sufficient for both understanding the process of negotiation of collective subjectivity in Virno and the politics of operations in Mezzadra and Neilson. This, however, is only possible to a degree in which the system's governance assumes the distributed character. As the system becomes decentralised, the pull of its momentum decreases, and it becomes more suited for scalability. Traction in a distributed system is higher in the sense that the system remains fully auditable regardless of its scale and complexity, according to the scale-free principle discussed in Chapter 5. Concomitantly, the more traction there is, the more prominently production practice can adopt diffractive behaviour. While such a presentation of labour may explain some aspects of the labour process in software capitalism, it is now important to turn to its other principle, which warrants the circulation of knowledge throughout the production lifecycle. It is the principle of the emergence of organisational culture which creates the conditions for enacting and reproducing labour relations in the first place.

The culture of software production within the organisational space

When referring to culture, the present thesis is generally interested in the *organisational culture of high-complexity production* as the body of knowledge shared within an organisation that helps weave the organisation's collectivity. To clarify such a definition, two points are pertinent: what kinds of shared knowledge are at stake, and why should the culture of high complexity regime be different from the one of

---

[87] Skelton and Pais, 2019: Ch.7.

low complexity. It has to be noted that some notions dealing with the category of shared knowledge have to be left out of the scope of current research, to be able to use a specific meaning in the subsequent discussion of how such knowledge is implicated with the software production lifecycle. Such related terms are *general intellect, institutional memory* and *group mind*. To decide on the required meaning of the term *culture* in my study, it is therefore necessary to understand in which way it might be different from these notions.

Production of shared knowledge. The *general intellect* is a concept developed in Italian Autonomist thought, which portrays knowledge as fixed capital that is no longer uniquely embedded in machinery or other physical assets but instead is included in the living labour of service workers. In this sense, the general intellect is something which is not necessarily attributed to any specific product, organisation or community of practice. The term *institutional memory,* used in some of the organisation theory and operations research, refers to knowledge which is found in the minds of the team members, and which the organisation aims to convert to a tangible, explicit form as the body of its documentation.[88] The present research uses a notion which is close to it when dealing with a part of the production lifecycle which contains a set of executable instructions for continuously deploying the software system – an *epistemic infrastructure as code* (EIAC). EIAC in this sense stands for the institutional memory, as much as the compiled code binary in runtime – that is, when the application is running – stands for the outcome of production. The third term, a *group mind* of an organisation, emphasises general knowledge as part of the behaviour of an organised human group. Herbert Simon notes that the group mind notion can be slightly misleading, in that often what appears as a group mind reveals itself upon closer inspection as an enactment of the various organisation's entities and relations, such as policies, balances of interests, loyalties of different involved parties, effects of employee training and authority links.[89] Rather than being a kind of self-organisation, such a group mind functions by the adherence to audit practices subjected to authority. In simpler organisational forms, each individual independently evaluates how their actions will resonate with the rest of the group. In complex systems, they coordinate with a general rule which acts as a predetermined course of action that would contribute to the corresponding criteria in a reliable way.[90]

Seen in this way, the terms such as *general intellect, institutional memory* or *group mind* would not capture the specific meaning of shared knowledge which is at stake in my research. It therefore, becomes necessary

---

88 Simon, 1997: 218 and Kim et al., 2016: 67.

89 Simon, 1997: 177.

90 Ibid.: 178.

to turn to the use of *organisational culture* as a term that would address a specific set of norms and assumptions which encompass a more-than-human capacity for knowing, can exist in a tacit form resistant to audit, and overlap the institutional boundaries to traverse both the organisations and the communities of practice (CoP). The latter facet becomes particularly important for the discussion of distributed collectivities in Chapter 4, where the notion of CoP is approached through the foundational theorisation of educational theorist Étienne Wenger. Wenger explains CoP as a form of mutual engagement with a particular problem or a joint enterprise of negotiating what the problem is, and in my research, I often expand this definition to everyone who contributes to the process of production of a software system, in contrast to production teams specifically formed within organisations.

Furthermore, attention to CoP makes it possible to grasp the specificity of shared knowledge that pertains specifically to production within the organisational sphere. This cultural segment is characterised by its adherence to the administrative protocols of governance, as well as the bottom-up initiatives within the wider CoP. The difference lies in the motivations of its members and depends on freedom and loyalty, which become particularly important in moments of change in the context of high complexity. On the one hand, there is an organisational tendency to develop ways to warrant the system's stability under the pressure to change and to preserve the continuities of its interrelations. On the other hand, there is a web of creative and informal daily cooperations between the members of the collective which ensure cohesion, albeit at the expense of formality of interactions and administrative protocol. The notion of shared knowledge for the present research, therefore, necessarily has to account for its continuous negotiation between these two spheres. It should account for the factor pointed out by Simon, that sanctions and rewards alone are not sufficient to advance the organisation's goals and there always has to be some form of enthusiasm for creativity and loyalty to the organisation's assumptions, practices and ethical norms.[91]

Management before and after software. Turning to the second cultural aspect, the question can be posed more specifically, why is the organisational culture in the high-complexity production context different from pre-software industrial low-complexity manufacturing? The hypothesis here, as the General Introduction has sketched out, is that in both high and low-complexity types, the stakeholders are present simultaneously in their empirical and epistemic capacities. Empirically, there are the *workers*, who bring to the table their labour-power, and the *capitalists* who own the means of production and distribution, and who alienate the results of workers' labour. Within the epistemic dimension, the equivalent of workers and capitalists are the *stakeholders* who do not have an antagonistic relationship, due to

---

[91] Simon, 1997: 21.

the absence of a requirement of mutual dependency for the sake of reproduction. They are instead a category of individuals or groups who could be potentially affected by the changes introduced into the system and can appear as business owners, production teams or the systems' users, or a combination of these. The stakeholders inherit their non-hierarchical quality from the larger category of agents, the autonomous entities of various kinds which comprise a unity within a system. In the definition of Péter Érdi we saw in the General Introduction, the agents are endowed with equal authority and are relatively independent in their capacity to choose the most appropriate strategy.[92]

While there is no particular reason to invalidate either the empirical divide based on the ownership of the means of production, or the latter epistemic view in an either high or low-complexity production context, the agent-based paradigm usually applies to the issues arising in complex systems. The consequences of grasping the organisational culture which is at stake in the present research, therefore, should be explained by the qualitative shift that had historically occurred in the empirical and epistemic categories in response to the change of labour composition from primarily industrial to primarily knowledge-based production. Combining Thrift's findings with what we have learnt from the discussion of the performative specificity of labour in software production in the previous section, the key differences are twofold. For one, the work outcomes are no longer as clear as to allow to rigidly plan and necessitate continuous negotiations between the stakeholders. Concomitantly, there is an increase in the managerial population of companies and the diversification of their functions, most of which are performative.

The pre-software paradigm, as we saw in Thrift earlier, occupied Western organisational thinking to a different degree throughout the whole of the 20th century, completely dissipating by the 1990s. The paradigm largely assumed management as an organisational layer primarily concerned with overseeing assembly-line algorithmic production and implementing the management based on metrics according to the Taylorist scientific management. It was shaped by the world of business dominated by large hierarchical and multidivisional corporations, which pursued a goal of increasing in scale by following a unified strategy.[93] Such a view of the firm, viewed in the context of the period's high cost of computation, explains the ubiquitous figure of an *organisation man*, a corporate employee controlled by the techno-scientific administration. The historian Lewis Mumford sees the organisation man as nothing more than a human functionary who stands in place of a computational entity and only acts as a temporary replacement for it. The figure is enacted, to Mumford, by a 'part of the human personality

---

92 Boisot in Boisot et al., 2007: 8.

93 Thrift, 2005: 31.

whose further potentialities for life and growth have been suppressed' for the larger purpose of a mechanically operated collectivity, where the model for the employee is the machine itself.[94] Due to such automatism, it is only natural that the increased computational capabilities made the automatic parts of human activity fall back into the sphere of computation. For example, the mundane clerkly routines of the 19th-century bureaucracy portrayed in Nikolay Gogol's short story, *Overcoat* (1842) are replaced by the office printers in the 20th century, which are, in turn, rendered obsolete by the algorithmically initiated, yet no less mundane, database backups in the 21st century.

In Thrift, the new capitalist paradigm began to evolve at the same time as the knowledge society. The latter presented the former via such challenges as the growth of information to be processed, the introduction of new market players and the increased speed of communication and logistics.[95] All of these, and many other changes in society and technology, might be seen in the context of the present study as underpinned by a larger trend of the falling cost of hardware and computation. The start of this trend, as we saw earlier, was announced by Gordon Moore in 1965, and it is likely to be a lasting tendency of the IT industry for the reasons further discussed in Chapter 5. At this moment, it is necessary to mention that because of its disruptive exponential character, the initial phase of computational depreciation caused an aggressive infiltration of software into all aspects of production and distribution. This created the context of high-complexity production, since every company that introduced the use of software, enabled the computational abstraction layering in its operations.

This change had two important consequences that made the organisational culture of high-complexity production different from its low-complexity counterpart. On the one hand, the centralised governance of technological systems could no longer provide sufficient traction, which resulted in software crises, such as the one described by Frederick Brooks, and created a shift towards complex adaptive production systems with the proliferation of situated and tacit knowledges. On the other hand, the capitalist formation had discovered the phenomenon of software complexity with its potentially limitless topologies of the problem spaces of production, and the possibility to invite a larger managerial population into the production process. The new managers, however, were no longer required as the functionaries, since those were being replaced by increasingly ubiquitous computation, but as performative workers that would be able to partake in negotiations engaging with the various types of tacit and explicit knowledges, for which purpose the transformation of culture in the organisation space was inevitable. Describing the preconditions of the production paradigm shift in this way helps to clarify the

---

[94] Mumford, 1970: 277-78.

[95] Thrift, 2005: 32.

second question of organisational culture about the distinctive features it acquires in the high-complexity production context. Yet, the discussion above draws attention to the last important aspect of software capitalism, the new pathways of valorisation that the paradigm shift has opened up, which is now necessary to discuss.

Computation, valorisation and culture

Borrowing the definition of valorisation from Marx's *Capital*, the present thesis understands it as a process through which capital increases the proportion of surplus value or profit in its overall revenue. According to Marx, in valorisation, capital increases its value 'through the unity of the labour process and the process of production of increased value.'[96] While the process of simply creating value happens at the moment of payment for the labour-power used in a specific event of production, valorisation creates value at any point in time after that moment.[97] While a more detailed analysis of the specific forms of valorisation of computation is a matter of a larger future research trajectory, the present study provides the context for some of the valorisation mechanisms specific to high-complexity production, as they appear throughout the study. These are: the resale of the depreciating computation, briefly touched upon in this section; the valorisation of the continuity of dysfunction based on the expanding of production, explained in Chapter 4; and the retention of tacit knowledge, which is partially addressed in Chapter 5 and the General Conclusion. The purpose of this section, which closes the discussion of the key features of software capitalism, is to sketch in broad strokes the general principles of circulation of value in its production process, conditioned by the ubiquitous computation, which is used by Thrift in his analysis of the engagement of capitalism with software, and which I see in the present context as the inevitable consequence of the tendency of the cost of computation to fall.

In Thrift, ubiquitous computation refers to the computation that is autonomous in that it does not require human involvement and is present instead as an infrastructural property of the production system.[98] This implies that in software capitalism organisations, teams and agents no longer engage with computation in the specific phases of production workflow, as was the case in the era of mainframe computers, but instead exist and interact within the production topology created and maintained computationally. Computation in this context stops being merely an operation that takes specific inputs and delivers a precise result but rather evolves to become something capable of presenting qualitative judgements and working with tentative, ambiguous propositions. As a result, it becomes possible to ab-

---

96 Marx, 1990: 36.

97 Ibid.: 302.

98 Thrift, 2005: 160.

stract the practicalities of creating the compiled code executed for the user in runtime and to suggest the software production system as a production system that simultaneously produces itself along with its products, operating through deployment and integration to stream the executable binary continuously. The computation-based space forms a *plane of immanence*, as Chapter 4 explains by adopting the term from the philosophy of Gilles Deleuze and Félix Guattari, which is a place of capitalist production components and their relations. Such space is porous enough to create DevOps as a superimposition of the methodologies of business operations and software development and to enable the continuous production design lifecycle. To achieve that, DevOps presents hardware, services, teams and other resources as abstractions available for symbolic manipulation. At the higher level of abstraction, the infrastructure itself is the code, which, as Chapter 3 demonstrates, appears in its deployment as a production of means of production, because it creates the entire topology of the production system, including all the required services, resources, a testing suite and the allocated virtual hardware.

As the ubiquity of computation becomes possible through the increase in computational capacity and emerges as an independent force, it initiates a tectonic shift in production relations that makes them less distinct from distribution and creates the conditions for a sphere of production culture, as we saw previously, as the space for emergent performativity of labour. Due to its radical non-humanity, ubiquitous computation is not connected to anyone's professional knowledge and takes place anywhere and out of context.[99] In Thrift's allegory, ubiquitous computation takes on a role of a new prosthetic layer of the real that carries out routine cognitive operations, not unlike a new force of nature. The calculations, he argues, 'are so numerous and so pervasive that they show up as forces rather than discrete operations.'[100] The outcome of such a mundane and unspecific presence of computation in the organisation's technology value stream is that additional computational resources are available at little or no additional cost, which makes the creation of surplus value through computational processing and transmission of knowledge a commonplace valorisation feature. One way of doing, it, which I dealt with in the field, is based on reselling the depreciating computation, which can be argued to lie at the core of cloud computing firms, such as Amazon Web Services (AWS). While AWS computation capacity is rented out, the value chain extends further into the AWS platform which is present as a service, providing an interface for managing the resources and various aspects of infrastructure. The service, in turn, is used by DevOps professionals who appear as the platform's community of practice. In this sense, the valorisation of computation is immediately linked to at least two kinds of services.[101]

---

[99] Thrift, 2005: 156.

[100] Thrift, 2008: 100.

[101] For my field engagement with DevOps professionals and AWS, see Appendix, CS21.

This culturally conditioned valorisation junction could benefit from the tangential critiques of capitalist modes of production and distribution. For example, in her theorisation of *supply chain capitalism*, the anthropologist Anna Tsing describes a model of commodity exchange that creates value via the extreme identification of production and distribution, typically realised through outsourcing or subcontracting labour. At the core of supply chain capitalism, Tsing argues, is the tendency of the workers and the managers to converge in the figure of a *servant leader.* Servant leadership celebrates the workers as new self-managing individual producers and decisively banishes from the economic realm the cultural aspects of their identity, such as gender, race, ethnicity, citizenship, age or sexuality. The characteristic double move of, on the one hand, erasing the legacy of labour struggles, and, on the other hand, encouraging the self-exploitation of workers, is made possible because the supply chain exchange mechanics remain reliably concealed among non-economic factors.[102] In real-life scenarios, the scheme is realised by presenting the workers as independent contractors who cannot help but keep the market rates low for everyone by pitching against one another, as in the examples of Fiverr and other contract-based platforms.

While the present phase of my project doesn't leave me enough space to deal with the cultural aspects of identity, the methods for concealing the value extraction and concomitant exploitation of workers are perceived here in alignment with the supply chain capitalism model. For example, due to the principle of retaining tacit knowledge, companies will be motivated to delay creating technical documentation because it is costly and does not yield immediate business value. Therefore, it would only be possible to alleviate the cognitive load of the workers when they are too overwhelmed or confused to be able to work without relevant technical documentation. Likewise, in exploiting Conway's Law, the management would always choose to increase the complexity of the company hierarchy to defer the decision-making in fact, concealing the creation of value by extending the value chain. Similarly to the value chain capitalism concept, the valorisation mechanisms are presented as organisational achievements. In the case of exploitation of tacit knowledge, there is the encouragement of a *Musketeer attitude*, which promotes mutual support among team members.[103] In the case of the deferral of dysfunction, there is a celebration of managerial outsourcing and delegation skills.[104]

A self-reinforcing valorisation model that emerges from such a presentation of software capitalism can be thus argued to engage with both the performative labour and the cultural sphere of high-complex-

---

[102] Tsing, 2009: 158.

[103] Rubin, 2012: 203.

[104] Conway, 1968: 31.

ity production. The two categories are employed in conjunction to mediate the extraction of surplus from the knowledge circulated within the problem space of production. Within the problem space of production, the three key criteria help to connect the three spheres, as the general schema of the production design lifecycle demonstrates (Fig. 4). Creating new requirements employs organisational forces, acceptance criteria engage with the technical system, and the customer value connects the market sphere.

Chapter conclusion

This chapter has outlined the context of my research, which is constituted by the three main traits in thinking about software production systems. First, systems theory offers two kinds of structuring of software production, centralised and distributed, with each having its benefits and downsides. The benefit of the centralised control is that it allows systems to rapidly produce great quantities of inventory, where the outcomes can be set out in advance and the workflow stays largely unchanged. At the same time, such systems are fragile in the face of change, with the main reason being that in such systems the relevance of methods is measured by their proximity to the central control mechanism. The strength of the distributed systems, on the contrary, has been attributed to their resilience in the face of complexity through being able to develop local solutions and situated knowledges. In the second instance, the chapter has considered that the popularity of the Agile methodology may be due to its ability to distribute the cognitive load and approach complexity from a topological perspective. This, in turn, suggests that software crises can be abstracted away by methodologically splitting their various problematics from the maintenance of the organisations' technology value stream. Lastly, the chapter has turned to the realisation of software capitalist value exchange via the organisational culture and the computational topological stratum. Here, the complexity has been explained in its role within a system to be the tool for valorisation that employs the system's dysfunction as the stimulus for the further expansion of production activities. The chapter has described such a valorisation mechanism as a defining characteristic of a current facet of the mode of software capitalist mode of production.

In the context established through reviewing the literature sources, there can be noticed an association of the process of change in technical systems with the activities of agents, which may point to the possibility that it is due to such activity that software production is capable of absorbing surplus amounts of living labour at rates higher than industrial manufacturing and other sectors of production. This absorption, however, only leads to further increases in complexity, which acts as a roadblock to the organisation's practices of audit, while promoting capital circulation. The agenda for the next chapters is therefore twofold – on the one hand, the argument needs to trace the infrastructural effects of software complexity to understand the negotiation of problems that happen among stakeholders, and in which

ways the complexity is circumvented in real-world software production scenarios. On the other hand, there is an urgency to sketch the source of complexity arising from concrete practices, primarily seen here as the effects of the falling costs of computation. The latter causes the misbalance between the organisation and the market, which in turn creates the continuity of dysfunction within the system in the face of software complexity. Studying such an elusive and ever-moving target makes use of the production system design lifecycle model, which the next chapter turns to, borrowing from the compositional framework, queer theory and abductive logical reasoning to create such a model.

Chapter 2. The compositional method

As Chapter 1 has aimed to demonstrate, highly complex production systems with a potentially unlimited number of components present the research with challenges different from the systems where the number of elements is finite and can be fully accounted for. This calls for developing a distinct study method that would see software systems as evolving through their complexity effects. This chapter works towards such a method by combining the elements of the three methodological frameworks. The first one is compositional methodology, a framework that sees the problems not as something externally imposed on the system, but instead as its integral parts. Viewed as such, problems act as communication vehicles for sharing meanings through the processes of continuous negotiations by the involved parties, such as business owners, agents of production or end users. The problems appear differently throughout the different stages of negotiations, yet are always associated with various kinds of knowledge inventory. Methodologically, such inventory can be split into *new knowledge* to be assimilated into the system and *confirmed knowledge*, verified in a way that makes it useful as grounds for further interpretations. This difference is reflected in the methodology as a division of the production lifecycle into the problem space and the epistemic infrastructure, which deal with two types of knowledge, respectively. The epistemic infrastructure appears as the suite of principles for organising what is known about the problem. The problem space is a domain of possibility of solutions that emerges in response to the application of the method, while also accounting for the change that the problem itself undergoes as it is being solved.

The second methodological motif is concerned with the aspect of verification of evidence, or objectivity, which I argue should be made possible through engaging with queer theory, and more specifically, diffraction. Diffraction here is seen as something that opens an escape route for the disadvantaged reflexive position and, instead of fixing the researcher's point of view, creates a dynamic multiplicity of viewpoints, which allows engaging with operations and software development in a queer way, by tracing the effects of difference. Here the queerness of the approach does not necessarily entail a queer reading of DevOps but rather employing the queer potential of transforming the existing relations through enabling different points of access, such as body performativity and affect. The third methodological constituent of my study deals specifically with constructing the production model in a situation of high uncertainty. This is done with abduction, as it is theorised in the foundational work of logician C.S. Peirce and the recent post-Peircean thought. Abductive modelling works through the tentative

propositions, preserving ignorance where an opportunity of deciding reliably for the long term does not present itself.

Composing problems

In its formal definition, a *compositional methodology* (CM) is the study of the possibilities that a problem has within a problem space. Beyond composing or putting things together, CM makes it possible to examine the *action* of putting together through the things it puts together, for example by looking at how things change through the application of the method. The topological understanding of the problem spaces in the present study is based on the fundamental theorisation of the philosophical notion of the plane of immanence, which is here applied more specifically to the sphere of production to present it as the plane of components and relations of production as well as their organising principles. The immanent view of the capitalist mode of production is viewed here through the lens of the philosophers Gilles Deleuze and Felix Guattari's presentation of capitalism as the formation that has no exterior limit, but only the interior limit that is continuously reproduced through its expansion. The capitalist general principle is immanent because it has its internal coherence which, rather than recognising the external world, works by setting up boundaries for the application of its own rules and extends the area of application of such rules, subsuming the external world by making it compliant.[105] In Chapter 4, the philosophical underpinning of the plane of immanence will be explored further to understand the capitalist principle of continuity of dysfunction, which will be argued to animate the circuit of the software production model.

Two further considerations face CM in the context of the practices of audit and any implications to governance. On the one hand, CM plays an active role in setting up the conditions of the topological unfolding of problem spaces, rather than their passive description.[106] On the other hand, while enacting their becoming topological, as the cultural theorist Celia Lury has it, the method also needs to account for how the conditions allow the reproduction of these spaces – that is, to understand the rules of its repeatability, which will allow for the practices of audit to take place. Such repeatability is crucial for audit because to undertake the review, the audit needs to understand how the problem composition is established through repeatable routines. The present study takes advantage of the topological capacity of CM and formulates the ideal shape of the knowledge-based capitalist value model with the software system at the centre of an organisation, in the same way DevOps research does it, with the aim to understand the possible consequences of such placement. This, in turn, suggests that some of the

---

[105] Deleuze and Guattari, 1983: 230.

[106] Lury, 2021: 9.

topological characteristics of the production model have to be prioritised, for example, the organisation design pattern that divides the teams into the ones tangential to direct provision of business value and those concerned with the specialised service and support work. Chapter 3 explains this pattern as stream-aligned team topology and suggests that problem spaces can be seen as emerging in the boundaries and neighbourhoods created by the intersections of the stream-aligned teams and non-aligned services. Such an approach to team design views the problem in compositional terms – that is, as Lury describes it, through the relational transformation of the context, without treating the problem space of production as a mere representation of the problem, in the sense of its being a re-presentation of the external referent.[107]

CM recognises the constituent parts out of which the problem space is composed, and provides the avenues for thinking about the double move of methods – in the first instance, their capacity to constitute the problem space, but in the second, to describe a circular motion in the development of the problems themselves. In Fig. 4, the production design lifecycle demonstrates the circulation of knowledge in terms of CM, among its two major constituent parts, the epistemic infrastructure as code (EIAC) and the problem space of production. For the present dissertation, the diagram serves as a methodological blueprint and guides the discussion of the parts and processes of the software design lifecycle. The process of deployment, defined here as making the composed things available to stakeholders, is not something fixed and numeric – albeit algorithmic and auditable – but rather a matter of distribution through bringing the executable code, the database, the environment and all the configurations into one place. The software system in the production circuit comes as a result of deployment as the symbolic conversion of the components, configuration and strategy as code into a concrete manifestation in the shape of the working digital product.

---

[107] Lury, 2021: 208.

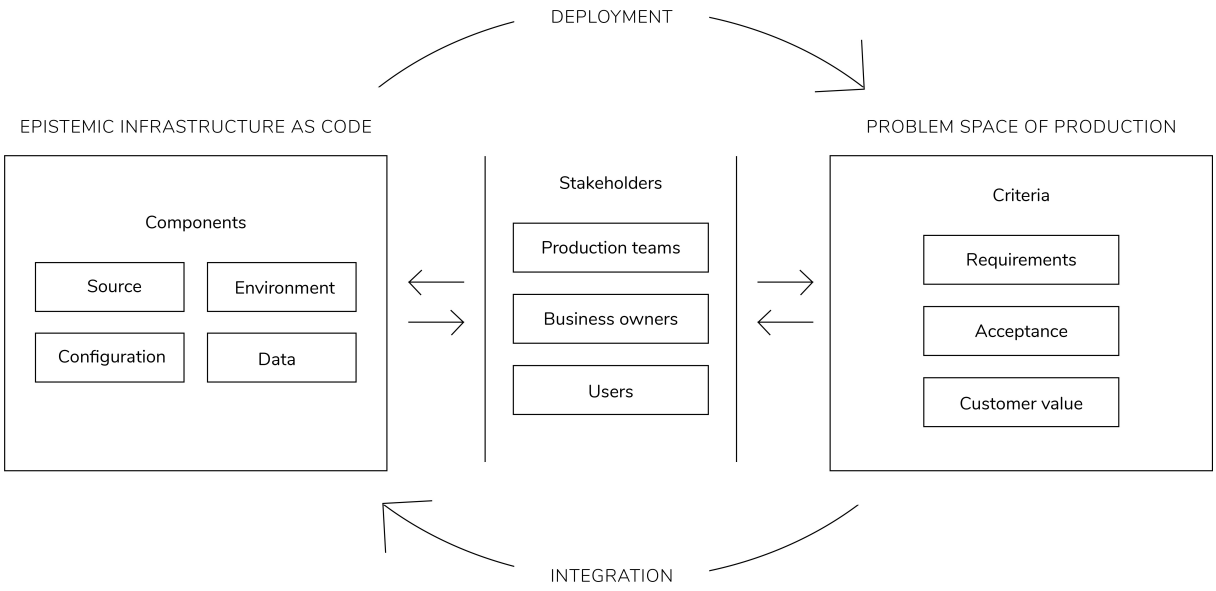

Fig. 4. Production design lifecycle.

Once deployed, the software appears in the problem space of production, where it comes into contact with the criteria, split here into three groups – requirements, acceptance, and customer value. Such encounter initiates the process of negotiation of the system among the stakeholder groups. As agreements are reached among the stakeholders, the meanings become assimilated in the form of knowledge throughout the agents and agent groups, and undergo the reverse symbolic conversion into code that adds to what is known about the problem into the EIAC ready for the next round of deployment. The two processes share their ongoing character with the two paradigms of DevOps – Continuous Deployment and Continuous Integration, which are becoming increasingly dominant in software production as the costs of computation continue to fall and the streaming of large quantities of data becomes more affordable. As noted above, the interpretation of knowledge in CM is open-ended in that it does not presume the problem as a given, but rather works with methods and facts to analyse the character of transformations that occur in the problem space to construct an appropriate way of approaching the facts about a specific case.

With this in mind, the construction of a production model is a fruitful exercise because it acts as an epistemic mediator to use the interpretation of a problem in terms of its becoming to think about its potentialities. Abductive manipulation, in the way the philosopher of science Lorenzo Magnani uses this method, becomes possible as a way of creating variations of the model to come to some strategic decisions, where the future criteria are unknown.[108] In this context, only the knowledge contained in

[108] Magnani, 2009: 35–36.

the EIAC can be treated as a reliable reference. The act of *composition* refers not only to the activities carried out by the researchers, but also the operations that are associated with the epistemic infrastructure, which take the shape of propositions, engagements, activations or, as is the case of EIAC examined in Chapter 3, automation.[109] The composition as a method is important because it permits focusing on the components in terms of their non-equivalence and spares the effort of taking account of the identical parts, which, as we saw, make no structural difference in software systems. The non-equivalence between components, however, can be infinitely extensive, due to the frequent use of incompatible technologies and paradigms, and therefore is usually resolved through adding complexity – a process which in turn, activates the valorisation mechanisms in the market domain.

CM enables studying the problem space by substituting the static notion of a problem with a process of problematisation. The process involves abductive manipulation that provides a way of reasoning about uncertain outcomes through cognitive and epistemic mediation of external models.[110] This in turn leads to the automatic production of a problem space, which means that the knowledge about the problem variants, produced through the mediations, accompanies the establishment of new rules of engagement that necessarily and automatically assume mutual topological relationships. While this means that every act of the application of the method contributes to the expansion of the problem space, the results of different applications do not merely add up to one another. The research is rather boundary-making, and as per the cutting together-apart principle of a diffractive method, it is when different things are brought together that the boundaries become most prominent.[111] Any relation bears the dynamic aspects that detect and activate the conflicts of interpretations, to be able to draw such boundaries. The boundaries in CM are important because they serve as the markers of understanding the emergent performative subjectivities, which appear differently in each new event of production in response to the change, momentum and control parameters of the production system.

Composition and ANT

Since the present study owes much of its methodological thinking to the group of intellectual practices known as *actor-network theory* (ANT), a brief explanation is required of how the former framework is used for my project. More precisely, ANT here is not seen exactly as a theory or method, but rather, as a way of thinking about research which benefits from the critiques developed by the original ANT thinkers themselves. Since its formation in the 1980s, ANT had been concerned with eliminating the

---

[109] Lury, 2021: 17.

[110] Magnani, 2009: X.

[111] Barad, 2007: 93.

distinctions between technical, social, economic or political factors, up to the point where a distinction can no longer be made between natural and cultural phenomena. ANT is characterised by one of its key proponents, the sociologist John Law, via its 'ruthless appliance of semiotics' in the sense that any entities only appear in specific capacities only through their associations with other entities.[112] Things also cannot be categorised before their relations are clarified because before that moment they don't have any inherent qualities. This has an important outcome in that ANT does away with various well-known dualisms such as human/non-human, true/false, and micro/macro, seeing them instead as network effects.

In a more recent development in ANT, which is sometimes referred to as post-ANT, there is a stronger emphasis on self-reflexivity, which sees ANT reflecting on itself. Such self-reflection is expressed via the multiplicity, fractality and complexity of the phenomena under consideration. Multiplicity means that there may be several versions of the same phenomenon, such as the instance of the same illness in different bodies that figure in the case studies of the sociologist Annemarie Mol,[113] or deployment of the same software system components in different environments, which creates the portable clusters of agents and relations that allow focusing the attention on the effects of difference. The fractality parameter implies that the versions of the phenomenon may be related, but not on all points or in all dimensions. In this sense, an example of a coastline drawn by Melanie Mitchell can be a good illustration of the fractal. The three images of the coastline – as viewed on the satellite image, observed from the nearby hill, and seen from up close when standing on the shore – appear similarly as rugged lines. In the first instance, the coast is rugged on a large scale, with inlets, bays or peninsulas; in the second and third cases, the ruggedness is still present but consists of the elements of a smaller scale.[114] Such an effect of the self-similar structure is what fractality refers to in the case of ANT – each of the components in the current actor-network conjunction can be exploded to reveal a self-similar structure within. In other words, self-reflexivity extends as a horizontal and a vertical series: any one given instance is self-same to the series of itself on the same level of abstraction, and simultaneously to the series extending downward, the instances that can be found inside it, and upward, the ones it is a constituent part of.

In real-life scenarios, multiplicity and fractality co-exist in various entanglements, which creates complexity as a third dimension of this structure, and concomitantly, as Annemarie Mol emphasises, com-

---

[112] Law in Law and Hassard, 1999: 3.

[113] Mol in Law and Mol, 2002: 247.

[114] Mitchell, 2009: 103.

plexity is the third major concern of ANT.[115] Due to the multiple and fractal specificities of the examined phenomena, they are rife with local knowledge, which concentrates on one instance of a series, yet may not be relevant to other instances of the series. The specificity is tactical, that is, no one instance is a complete replica of another one, and yet strategically the parts of the system adhere to the general pattern informed by the production systems' epistemic infrastructure. Hereby, the present research links the fractal principle to the auditability requirement of the production context, seeing the self-sameness as the hope for avoiding the confrontation with the full force of complexity which would eradicate any possibility for planning and review. Similarly, to be consistent, it seems plausible to generally refer to the entities that the system consists of as agents rather than actors. This is possible because the term *agent* can be used to refer to the scalable analytical unit, which can be kept consistent with the current abstraction layer, and therefore can be flexibly adjusted according to the fractality principle.

In alignment with the post-ANT critique, I see ANT as a framework, but not as a theory or method. In the first instance, ANT can only be considered a theory in a minimal sense due to its design, which is meant to be theory-agnostic. Second, ANT is not meant to be a method, since its 'ruthless semiotics' does not let ANT make any specific recipes as to what the researcher has to do. Instead, as the sociologists Christopher Gad and Casper Bruun Jensen explain, the implication is that relevant actors, and what it is that comprises a network, can only be determined based on the understanding of the practices, the relevant local categories and differences. This makes it impossible to carry any assumptions outside of the field, before research takes place, or offer any concrete methodological propositions outside of specific cases.[116] The only motif that relates it to any method is that it reminds the researchers that whenever they go, they are likely to find clusterings or 'hybrids of action' comprised of actor-network relations, rather than any definite entities. In other words, ANT, to Gad and Jensen, gains value when it forms specific constellations with matters of empirical study,[117] creating the interpretation of knowledge as practice.

As an upshot, this thesis does borrow some specific ideas from ANT, such as fractality, which is utilised in Chapter 5 to understand scale-free systems. In the present research, I am less interested in computing or management as symbolic manipulation and rather focus on developing a workable model for abductive analysis. For this purpose, it is important to understand how knowledge is assimilated and valorised through nested problem spaces of various applications of tentative propositions, which makes

---

[115] Mol in Law and Mol, 2002: 246.

[116] Gad and Jensen, 2010: 76.

[117] Ibid.: 75.

it more urgent to think not in terms of actors and networks, but rather in terms of agents and systems. Agents are primarily used here to be able to describe the principles of stakeholder involvement with the feedback criteria of the problem space, and the notion of the system as a topological unity is employed to grasp the mechanisms of boundary-making practices. Furthermore, it is in the project's best interests to follow a methodological schema proposed by operations research in IT or DevOps, which itself is already a hybrid discipline emerging from operations and development methodologies, to be able to comprehend the tangible qualities of frameworks it creates. In this sense, rather than adding to this recipe, ANT works as a robust set of guidelines without the obligation to aim for a description of their precise configurations as a research objective.

Goals and patterns of the problem space

Providing some further details on the notion of the problem space at this point seems necessary. As a general working definition for this research, the problem space is the space in which the search for solutions takes place, alongside the iterative process of articulation of the problem itself. The early recognition of the importance of the problem space as a problem-solving device belongs to Herbert Simon, who suggested that 'every problem-solving effort must begin with creating a representation of the problem—a problem space in which the search for the solution can take place.'[118] This space contains the conditions of possibility for solutions, largely avoiding carrying the assumptions or validating the problem with an external referent, such as through the evidence already contained in the epistemic infrastructure. Rather, since the solutions cannot be final and there can be no certainty about the premises, the best option is to work with a model for abductive manipulation. Such a model is created through an iterative selection of situations that allow the potential resolutions to evolve relative to the problem, which is also simultaneously undergoing transformation.[119] The two pertinent characteristics of the problem space that are necessary to mention here are its goal orientation and repeatability.

Goal orientation presupposes that the becoming of the problem space usually begins with a goal, which may often be tentative, yet the goal is important for initiating abductive modelling, through which then the goal is further adjusted. By convention, the goals are complemented in the problem space by the givens and operators (Fig. 5). Due to the changes that the three components undergo, the problem space, described in terms of relations between them, changes too, which makes the problems re-emerge in any moment where the system finds itself in misbalance. The *goals* can be defined as the definitions of done or conditions that have to be met for the problem to reach its desired end state.

---

[118] Simon, 2019: 108.

[119] Lury, 2021: 208.

The *givens* are the facts that describe the problem. And the *operators* are the actions that create the continuity between the givens and the goals.[120] I apply these three CM terms equally for software production, to acknowledge that the negotiation of the problematics in IT products is generally underpinned by a similar logic. The givens of software products evolve as new hardware or technical styles emerge through invention or innovation. The new goals emerge as the new releases of products get tested and rolled out, and the new use cases come back to production, opening up opportunities for optimising the system's technical functioning. Changes in givens and goals give rise to new operators, such as methods and concepts, which are used to update the definition of problems to situate them according to the changing requirements of the problem space.

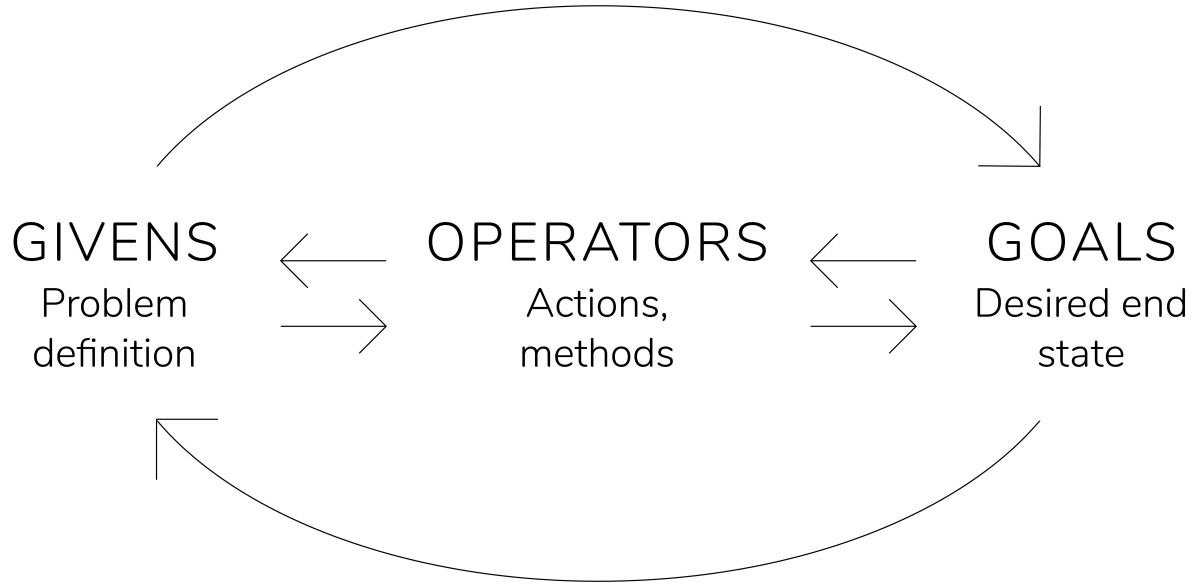

Fig. 5. Component relations in the problem space of production.

Importantly, the progression between the three components is not linear: the circulation, or the process of mutually abductive definition of problems and their spaces, can play different roles. Not only do the problems themselves have different capacities to employ the various properties of circulation, but the reverse is also true: the different forms of circulation support the explication of problems.[121] Goals can change due to market pressures or organisational changes, and can lead to changes in givens and operators. The change in the givens, such as rapid growth of computational capacity in software, can bring about organisational changes. The failure of organisations to respond promptly may lead to major failures, as in the case of historical crises in IBM in the 1950s. Yet, of course, the misbalances can be evoked by the changes in operators too, as in the examples of the Taylorist style of scientific manage-

[120] Lury, 2021: 2.

[121] Ibid.: 48.

ment, the emergence of the Human Relations segment of organisation studies, or the movements for refusal of work, such as those advocated by the Italian Autonomists.[122] As one of the examples later in this section will illustrate, a change request which may emerge among the stakeholder groups may not be a problem itself, but could rather be treated as a symptom, through which the composition of its nested problem space begins. A first draft of the actual problem can be formulated in the process of data collection, staff interviews and discussion of requirements.

While the whole of the problem space is too difficult to represent, the goal orientation makes it possible to create a visualisation, contrary to Frederick Brooks's software invisibility observation. Creating the problem in these conditions could rather be described via the analogy with the late 19th-century study of movement done using laboratory photography, where the recordings of movement phases were superimposed into a single image (Fig. 6). Likewise, the superimposition of the problem in its different moments of becoming onto the problem domain creates a range of inquiries into the problem space that does not aim at creating a complete external representation, but only the movements, as the image of the pelican shows, pertaining to a specific activity. The image distinguishes the movements of the pelican linked to flying without adding any other potential pelican movements, which would have created the excessive complexity of visualisation. The goal orientation in this sense provides a more detailed account of the problem by nesting the problem space within a larger problem topology terrain.

[122] For the in-depth critical discussion of Taylorism and Human Relations, Cf. Hanlon, 2016: 7. For Autonomia, Cf. Virno, 2004: 9 ff.

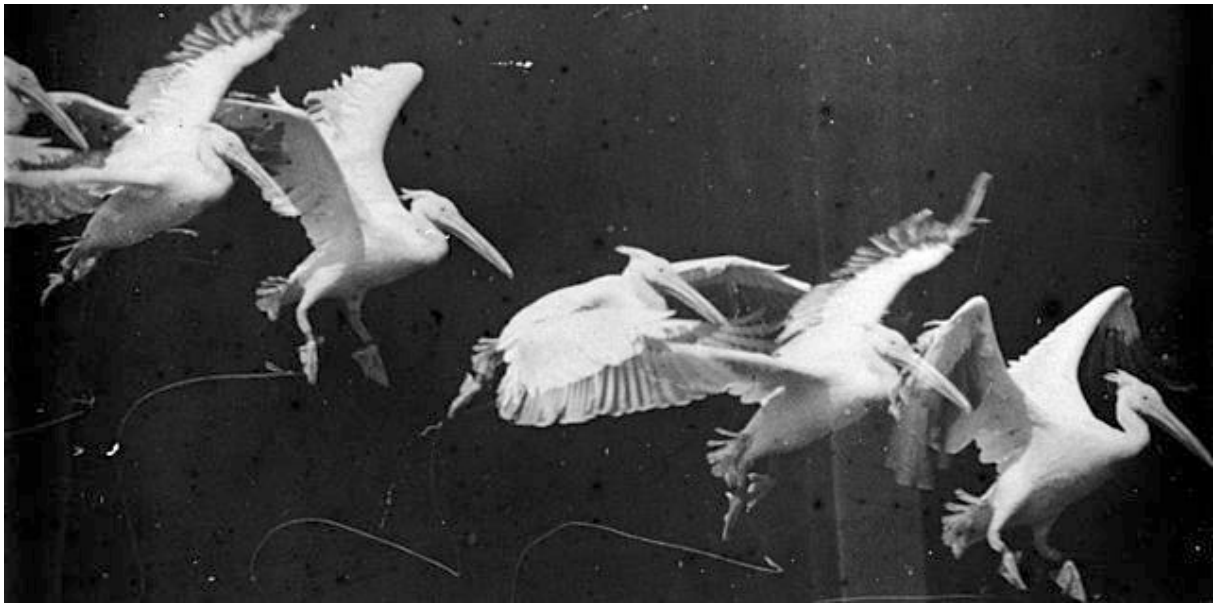

Fig. 6. E.J. Marey, A photo of a flying pelican, circa 1882. Source: Wikimedia. https://commons.wikimedia.org/wiki/File:Marey_-_birds.jpg

Since a problem does not disappear from the problem space once it has been addressed, but rather becomes one of its properties, it is implied that the terrain is largely constituted not only by the solutions to problems but by the past solutions and future problems, as well as the possibilities of the future solutions and the problems addressed before. Therefore, it is important to be able to retrace the steps, in case the problem arises again or a new instance of a problem case presents an opportunity to see why the previous solution is no longer relevant. Repeatability in this context is important for abductive manipulation because it allows adding a variety of states to the problem space, providing tentative, uncertain solutions to the problems that are not fully graspable simply for the reason they have not yet fully occurred. The repetitive movements within the problem space leave deeper traces and suggest the topological patterns which eventually transform into infrastructural features. Needless to add, repeatability is also important for making the system available for audit, since every repeatable procedure can be included as a standard in the project's technical documentation and version log, making it accessible to search engines or abbreviated for reports through the regular expressions.

Epistemology as the infrastructure of already known

Since any organisational change modifies the configuration of the problem space of production, stabilisation of what exactly constitutes the matter of research is the top priority. For this purpose, the becoming topological of the problem space becomes an indispensable consideration, since it would allow tracing the properties which remain invariant among the numerous transformations. Thus presented, the unexpected can still be reasonably accounted for, falling, as it does, within a certain territory that has some representation on the map of the overall terrain. Yet, dealing with problems purely as the en-

tities of a problem space would make the method appear detached from the reality of the domain. To compensate for this pitfall, the composition of the problem space of production has to go hand in hand with its *epistemic infrastructure*, which should be most appropriately defined in relation to the notion of epistemic culture in a theorisation of the anthropologist Karin Knorr-Cetina. To Knorr-Cetina, an epistemic culture has an infrastructural quality in its capacity as a suite of principles for organising knowledge, which serves as the means of the orientation of understandings, explanations, justification and beliefs about what is known rather than the content of the known itself.[123] Likewise, the epistemic here assumes an infrastructural quality because it provides material support for knowledge, which in software production systems usually comes in the form of technical documentation, production reports and company policies.

Despite the materiality of its being in the world, in the shape of documents, code binaries and databases, epistemic infrastructure does not necessarily imply that the knowledge it reflects and situates is fixed in place. Instead, the infrastructure as the idea of what is possible is iteratively composed together with the problem space, which is the space where the possible may take place. Yet, the application of any method does not only bring the outcome in terms of solving the problem but also in terms of the impact the act of problem-solving has on the problem itself. Such impact creates an adjustment in what is known about the problem, which is fed back via the integration movement into the epistemic infrastructure. To propose a method, the problem has to be put together first, yet the requirements that can never be written fully in advance make it impossible to outline the full idea of what the problem is, what it could be, or is going to be, and some part of the requirements is always located on the other side, on the yet-unsolved part of the problem. The process of composition thus is not something that comes from either the inside or the outside of the problem but rather is shaped across the space which emerges in the act of problem-solving.

As part of my field duties, I was often searching through the archive of Jira that our production team used for the past three years for technical support work. Looking through Jira tickets, I frequently found that many tickets from the past were either still open with requirements written up to varied degrees, or had to be reopened, as the issues that were addressed years ago had resurfaced once more. Some problems were associated with whole clusters of tickets, which gave a manifold presentation of previous work. The linked tickets often contained accounts of different aspects of the problem that had been dealt with by different staff members. The outcome of this archival component of my inquiry into the software system led me to conclude that the reason why the tickets on Jira are 'closed' rather

---

[123] Knorr-Cetina, 2007: 361, 363.

than 'deleted' when the work is finished is precisely that the problems do not vanish completely after they have been addressed. Rather, they transform into potential new givens or operators, partly merging into the ways of solving similar problems as they arise. With every new ticket opened, the system, as an epistemic infrastructure, moves closer to a more accurate account of the problem space of the product that it services.

Through working on cases like these, I became increasingly convinced that Jira is capable of providing a deep insight into the epistemic infrastructure of the organisation's software system and the workflow – in other words, a tool that is perfectly suited to learn about the organisation itself. In my fieldwork, Jira played a crucial role in capturing feature requirements, maintaining the acceptance criteria and evaluating the delivery of customer value, thereby encompassing all the aspects of the problem space of production. Importantly, these activities were happening not in a linear progression, but rather based on support tickets, which ensured that each piece of work was assessed in its context, and in relation to other work. As such, Jira warranted the terrain of the possibility for the emergence or re-emergence of problems because the documentation of the organisation's goals and budgets, as well as reports on required computational and human resources, could all be accessed together. Turning to a more detailed discussion of a support ticket in the problem space of production in a later chapter, I find evidence that the ticket is important both for the deployment and the integration procedures because it is located on the boundary between the business and the technology value streams, interfacing with both.

Studying the uncertain domain

Having seen the key principles of composing problems in the previous section, it is now time to turn to the practical way of modelling the production design lifecycle, in the context in which the problem composition takes place. It is important to recognise that such modelling is necessary when initiating case studies of a given production system and is also an ongoing procedure that is capable of recognising change and adjusting for new data as it comes in through the problem space. This section looks at the two approaches that this study's way of creating models benefits from – diffraction and abduction. Within a combination of these two methods, diffraction makes it possible to occupy a multiple and more-than-human notion of the system's operations. Here, diffraction is an opportunity to employ queer theory as a way of thinking differently, and including the body, performativity and affective dimensions in the study of DevOps. The other ingredient, abductive modelling, helps to preserve the ignorance while still being able to hypothesise, which is essential to the study of software as something that is rapidly changing and presents a constant risk of radical complexity increase, able to crush any approach which is excessively hard-bound to its underlying assumptions.

Queering DevOps

The risk faced by the technical DevOps literature, despite the adoption of such situated analytical frames as team topology, is to get entangled in a reflexive epistemology that maintains a sharp distinction between the software and the rest of the constituent parts of the production design lifecycle, with a routine emphasis on productivity. For example, technical innovations often occupy centre stage, giving little consideration to any of the political aspects of production, such as why the software is produced in the first place, or what kind of ethics are at stake. The section considers the two ways in which the relation of the researcher to the research can lead to a methodological rabbit hole, through exercising either an absolutely relativist or overly representationalist accounts. Absolute relativism leans towards studying relations without any attention to the role of the researcher, while the excessive prioritisation of representation, conversely, tends to rely on external evaluation of phenomena under study, and pays less attention to the relations, which places it at risk of missing the inconsistencies that superimposition of multiple viewpoints may reveal. At this point of discussion, I contend that the diffractive approach, inspired here by the thinking of the philosopher Karen Barad, provides a third view, which acknowledges the active participation of the researcher, without, however, the urgency for the situated knowledge to undergo an external validation. Both reflection and diffraction approaches, Barad explains, come from the definitions of the optical phenomena, yet differ in that the former predominantly deals with mirroring and sameness, while the latter 'attends to the patterns of difference.'[124] Returning to the model of a production system that the present study aims to create, the complexity effects are best identified by adopting a view from multiple vantage points, which could include communities of practice and the policies of specific organisations.

Diffraction and reflection. The term *diffraction* originally comes from natural sciences, where it denotes the distortion and interference of waves as they come into contact with other waves or encounter obstructions. To diffract, in its traditional use in optics, means to aim apart in different directions. Working in the intersection of particle physics and gender studies, philosopher Karen Barad shifts the principle into the taking together-apart of human and non-human phenomena by placing them into the plane of the symbolic-material becoming of matter. Diffraction is not the same as critique, and in fact, it's opposite because it works by affirmation rather than negation, and seeks to identify the patterns of interference which appear when things get taken apart and superimposed once again. As Barad puts it, 'diffractive reading might be understood as a form of affirmative engagement … A diffractive methodology seeks to work constructively and deconstructively (not destructively)…'[125] This, in

---

[124] Barad, 2007: 29 citing Haraway.

[125] Barad, 2014: 187.

Barad, makes it possible to identify previously inaccessible patterns of understanding and becoming of matters of negotiation. In a similar vein, the present study's application of diffraction is an attempt at *queering DevOps* which questions the reflexively normative vantage points which prescribe a specific representation of operations, without recognising the positions that are being negotiated throughout the software production process. The queer view makes it possible to grasp the uncertainty as a situation where no terms exist independently of the relation,[126] and the terms are only defined in their coming-together, rather than as a mere collection of moments of production captured by productivity metrics.

Breaking things apart is an essential constituent of the CM – or rather, to be able to compose, there has to be a diffractional viewpoint that recognises how the opposite qualities come together within.[127] For example, a diffraction pattern helps to visualise the connections among the political orientations of agents within the problem space and the associated epistemic infrastructure, which is not capable of political action of its own, and, consequently, does not counter or support any epistemological positions. *Diffraction* is a useful counterpoint to *reflection*: both phenomena come from optics, yet reflection is only a mirror image, caught up in the loop of finding the places of similarity. By contrast, diffraction is interference, or something that arises from intersections and differences, and therefore makes it possible to identify the differential patterns.[128] Focussing the research method on diffractive patterns exhibits the entangled and interactive processes of not only the modes of being but also the emergent nature of the modes of knowing. Therefore, it becomes evident that the study of the software production system cannot be taken as the study of any of its parts, deployment, integration, criteria or stakeholders.

Likewise, research that merely focuses on the system itself without simultaneous inquiry into the methods of its analysis is at risk of falling into some of the pre-existing hegemonic epistemologies. Instead, all parts of the system have to be taken together, as an apparatus of investigation. The notion of an *apparatus,* which has a complex history in various methodological schemas, is taken here in terms of its dynamism as a specific material configuration that goes beyond a mere static instrumental embodiment and is actively engaged in the iterative reconfigurations of research matters in their epistemological, political, and other sorts of becoming.[129] Because of such a character of involvement, the understanding of the phenomenon itself proceeds through cycles, where each preceding understanding en-

---

[126] Barad, 2011: 154.

[127] Barad, 2014: 174.

[128] Barad, 2007: 71.

[129] Ibid.: 142.

folds into the methods of research, which in turn results in sharper investigative tactics.[130] Similarly, the design lifecycle model at stake in the present research is not a mere means of passive accumulation of knowledge acquired in the field, but precisely a continuous process of *design of the production system*, through the entanglement of methods with the process of inquiry into operations. The cyclical motion entails deploying existing knowledge into the problem space, which in turn feeds back, through the process of integration, into the epistemic infrastructure, being actively involved in its re-articulation. This constitutes the extended understanding of the process of Continuous Deployment as described by its originator, DevOps practitioner David Farley, discussed further in the next chapter.

Risks of reflexivity. To understand where my research stands in relation to reflexivity, in its capacity as a method of constructing evidence through a systematic account of the role of the researcher, I think of its various manifestations through the discussion of the sociologist Steve Woolgar. In Woolgar, reflexivity is generally construed as an optic that uses the figure of the researcher as the means of variously splitting the research domain into the three areas: phenomena under study, the part that carries out the study – an individual researcher or a research body – and the matters of mediation with the mediating entities present as documents, but also as other human or non-human agents. For example, an object can be seen as an underlying reality, and the document is its surface appearance. Reflexivity usually relies on the adequacy of representation to consider either the distinction between representation and object, or the similarity between them.[131] Such a necessity to maintain or verify the relevance of representation to the object can lead to either of the two extremes.

In the first instance, it risks absolute relativism, which in essence has little to offer in terms of concrete research outcomes because it does not allow for any invariants, present in operations research in the form of standards, best practices, policies or regulations. In such a view, there is no longer an option to account for the social conditions of research in terms of the identity of the knowers, that are high on the agenda of the queer or feminist lens. Since the knowing subject is not in the picture, their identity, through their exclusion from the inquiry, gets substituted with assumptions. In the second instance, however, the reflexivity may go the opposite way and bring overly fixed ideas about the positions of the researched and the researcher. This might mean disqualifying any new knowledge that the inquiry may yield by giving too much priority to what is already known about the problem. The risk of such an approach is that dealing with representations, instead of the matters of research themselves, may make the problems difficult to detect due to the isolated nature of the models' system of reference. Given

---

[130] Barad, 2007: 73.

[131] Woolgar in Woolgar, 1991: 20.

such risks, the diffractive approach, as described above, can provide a third path which affirmatively engages with the figure of the researcher and considers the local practices on their own terms. The queer diffractive view, as Chapter 1 discusses in terms of its performativity, allows me to investigate the research through the act of cutting-suturing engagement with the entanglement of its differences.

Diffraction, dysfunction, DevOps. Reflexivity as a method for locating the researcher and organising the associated relations, therefore, does not imply any problems in and of itself but can lead to problems depending on its use. This can be illustrated with the example of Frederick Brooks's dilemma of software invisibility we saw in Chapter 1. In terms of reflexivity, Brooks's conceptual integrity schema places the researcher as an external referent, which makes it impossible to construct an adequate representation of the software production system, since, due to Brooks's own admission, it contains multiple diagrams at once.[132] The complexity of a software system appears as a conjunction of differences which stand in opposition to the uniformity demanded by Brooks's centralised administrative schema, forbidding a disinterested *view from nowhere* as reflection without practical or affective participation. The fact that software systems are essentially hard to understand makes them opaque, or as we saw previously, their essential queerness makes them appear differently depending on the context and reflexive research positioning. In this sense, the software invisibility appears as a dilemma to the conceptual integrity paradigm, yet it can be effectively modelled when approached diffractively from a diversity of vantage points. In this case, the fact that a system contains multiple diagrams at once, which is a problem to Brooks, would not appear as a problem to a production team using an approach which is open to diffraction, such as Agile.

Agile methods can, in large part, address the reflexivity dilemma through the disruption-based workflow, which may partly explain the popularity of this production methodology, at least in the granular day-to-day production activities. A key diffractive feature of Agile is that instead of having to bring into production the fixed references, such as rigid requirements or architectural blueprints to validate, it works by affirmation, evaluating the specific material entanglements directly.[133] Agile makes it possible to pursue the diffraction method's agenda to identify which differences matter, how and for whom, rather than becoming a *view from nowhere.* As Barad puts it, diffraction gets past the representations and instead offers a 'critical practice of engagement, not a distance-learning practice of reflecting from afar'.[134] In terms of its orientation towards disruption, as Chapter 4 will explain further when dis-

---

[132] Brooks, 1995: 185.

[133] Barad, 2007: 87.

[134] Ibid.: 90.

cussing the theorisation of desiring-production of Deleuze and Guattari, Agile makes it possible to engage with the production's epistemic infrastructure and problem space through the discrete parts evolving within the topology of the capitalist plane of immanence. Due to the discreteness of parts, Agile makes it possible to create a software system that would have dysfunction as a necessary precondition, induced both by the capitalist value relation and by the control society governance regime. The sphere of capitalist value relations requires continuous system breakdown to create technologies, processes and infrastructures to reproduce and scale itself through intensified circulation. The domain of governance, through its struggles to maintain control, creates the disruption-oriented environment of distributed audit, which uses complexity as a pre-condition of its fractal construction.

Thinking about DevOps with diffraction reveals that the two-slit point of view from development and operations enables a more direct and specific overlay of the topology of the software production system with the topology of business operations. The resulting combined terrain opens up the continuities that bridge the gaps between the technical components and the different parts of the knowledge and value circuits, allowing it to compress the repetitive acts via its ability to codify temporality into its infrastructural schemas. Furthermore, the intentional design of the organisation's communication lines, as per Conway's Law, along the architectural seams of the software system is effective enough to advance the required production model without the need of writing a precise list of always already outdated requirements beforehand. Instead, it renders total control unnecessary through the strategic endorsement of communication that produces desired organisational effects, characterised by either self-organisation or cooperation.

Creating abductive models

How is it possible to reason about the production model in the absence of a stable backdrop against which the decisions can be evaluated? The research logic suggests the two common ways of reasoning, *deduction* and *induction*, neither of which can, however, be relied upon in extreme uncertainty. Thinking with the originator of pragmatic reasoning about problems, Charles S. Peirce, deductive reasoning requires a certain organisational context to propose a theory that would form the basis of the research trajectory, against which the outcomes are evaluated. Inductive reasoning, in contrast, requires a rule to evaluate the cause and effect, usually by looking at individual interpretations that build up to form a generalised account.[135] Peirce appoints deduction as non-ampliative, in that it does not allow broadening by adding new information, and thus certain: given the premises are true, the results can also be guaranteed to be true. Reversely, induction is ampliative and uncertain, with results often requiring fur-

[135] Peirce, 1955: 152.

ther testing.[136] To go one step further and allow for hypotheses where neither the premises nor the outcomes can be fully relied on, Peirce offers a third way, *abduction*, which does not aim at warranting the truth, thereby making it possible to preserve the uncertain and unreliable knowledge in its uncertain form. Deduction and induction are the staple tools of any exercise in logic and are relatively low in their computational complexity. Abduction, on the other hand, may be more demanding because it is non-trivial and requires a leap of imagination from observation to understanding the intentions. Much relied on in real life, abduction has no ground for being valid from the strictly logical point of view, since instead of inductively arriving at solutions it affirms the consequent through guesswork. Abduction is the entertaining of a hypothesis, or a proposition, made without prior knowledge or testing, that an observed phenomenon may take place under different circumstances.[137] And since the initial proposals, specifically in strategic implementations of software production systems, are frequently in no position to carry out or access prior knowledge, abduction plays a central role in theory building.

Similarly to induction, abduction serves the purpose of expanding knowledge beyond observation, albeit in a different way. While induction infers about the future course of events based on what is known to have happened, abduction concerns the unobserved or the speculative causes of the observed, which gives way to the manipulation of events. What use can such an anti-logical way of thinking have in a research method? While it may strike the researcher as overly relaxed, abduction offers options that other, more logical tools do not. For example, that leap of faith opens a pathway for shifting from inductive to deductive reasoning within the bounds of one argument, which in turn makes it possible to iteratively build theoretical frameworks as they evolve together with empirical research and engagement with documentation and other references, something that makes abductive logic important in business analysis. Moreover, like induction, being ampliative, abduction approaches new data, often conflicting and contradictory, that comes in from field research, without any assumptions established beforehand.[138] In the formal tradition of case study research, abductive logic is commonly used to enable the researcher to explore the social phenomenon through the eyes of the social actors. This endeavour is, however, not any less risky in software research than in any other sociological and cultural studies: after all, the signification would not be the same in different organisations, even where the general context is similar, which makes it necessary for the researcher to rely on the signs without being sure what they mean. Abduction makes it possible to rapidly chart the problem space and to infer

---

[136] Peirce, 1955: 197.

[137] Ibid.: 152.

[138] Lury, 2021: 159.

based on what could be guessed about the reasons for events, rather than on the requirements written in advance.

As the analogic move that is characterised by its openness to exploration, creativity and curiosity, abduction fosters the interactions between the disciplines that may otherwise be kept separate. As Peirce notes, abduction makes it possible to expose the nature of the phenomenon by making inferences from the observed data or events to existing knowledge to understand their patterns. In Peirce's example, since in the abductive form of inference 'the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them,'[139] the reasoning would proceed in the following sequence: 'The surprising fact, C, is observed; but if A were true, C would be a matter of course; hence, there is reason to suspect that A is true.'[140] The repudiation of an inductive move in the inference introduces guesswork, which opens the argument to creative solutions. The situation of software production invites using such a type of inference on a routine basis. The reasons for that could vary, from the opacity of organisational communications or the gaps in institutional memory to the ambiguity of behaviour in users of the software. For example, some facts might have been communicated to the narrow circle of staff without informing all of the staff associated with them. The act of communication might have happened years ago, as is sometimes the case when dealing with system bugs that were discovered after a new release or upgrade. Institutional memory may also have ruptures, which makes current staff ponder about some decisions and past solutions, which may have been left without proper documentation and thus require a leap of faith to imagine what they were.

As a scant-resource strategy, abduction acts as a response to the ignorance problem – a situation where the cognitive target cannot be reached with current knowledge. The usual ways of dealing with ignorance are either by attaining additional knowledge, by making peace with the absence of knowledge, even if temporarily, or, lastly, by employing abduction.[141] In the latter, the agent has the grounds for action, even without giving any assurance or evidence that the ignorance is or is going to be overcome. One of the benefits, and, as Peirce contends, the inevitable effect of abduction, is the emergence of models and diagrams. Peirce ascribes a great deal of importance to diagrammatic thinking, which presupposes that the phenomena are constituted by the mind in a creative and model-based way. In other words, there are essential and indispensable perceptual presentations, that are presented to the mind's eye and cause the bodily responses to the phenomena which are not yet present in the external reality,

---

[139] Peirce, 1955: 151.

[140] Ibid.: 151.

[141] Magnani, 2009: 65.

to be able to derive any preliminary ideas of what the actual perceptual encounters might be.[142] The philosopher of science Lorenzo Magnani explains that while most of such affective diagrammatic inferences are a part of routine cognitive behaviour and do not have much importance, they can also cause significant bodily responses, and all of them should be understood as valid as the scientifically employed abductive inferences of any other models. The model-based abduction is understood by Magnani as extratheoretical, or opposed to theory-based abduction, and arises from the fast and uncontrolled knowledge-producing function of the body. The perception here appears as a tool for rapidly retrieving and grouping knowledge previously organised by different more long-term inferential processes, producing ideas which are so seamless and habitual that they appear as matters of fact.[143]

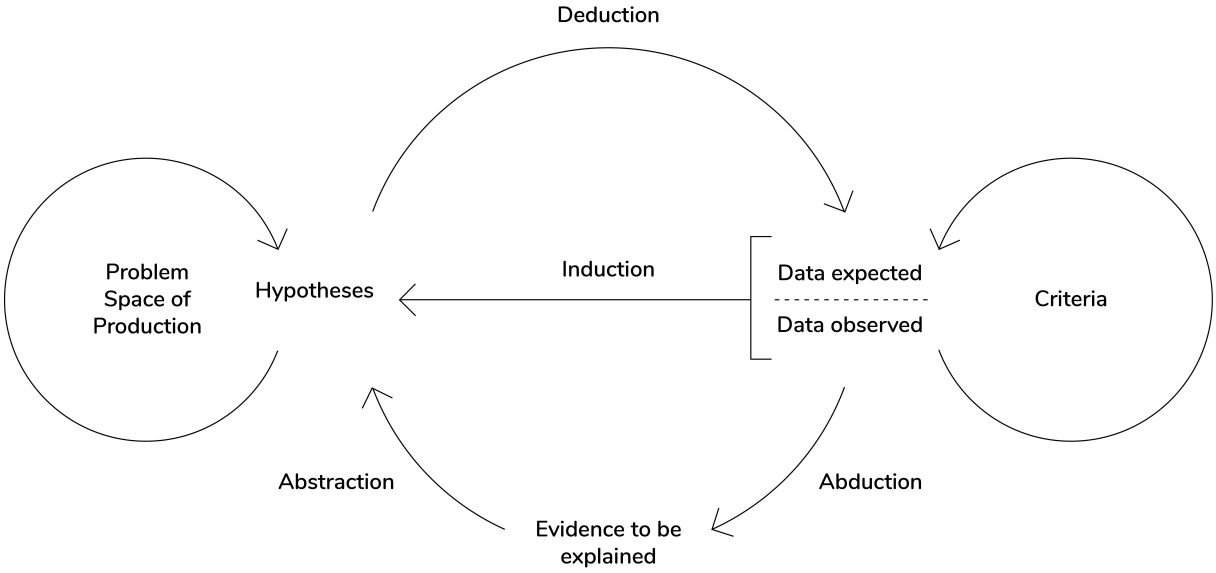

Fig. 7. The abductive problem negotiation event. Adapted from: Magnani, 2009: 16.

A general schema of abductive reasoning that happens throughout the negotiation of meaning in the problem space of production can be adopted from Magnani as shown in Fig. 7. The diagram indicates abductive inference as a relation between the problem space of production and the incoming data, such as requirements, the definitions of done and the criteria of customer value. The circuit is animated through the iterative steps of knowing, selecting and observing. Thus, the new data incoming on the right-hand side of the diagram is used to create the plausible diagnostic hypotheses that get inductively injected into the problem space, where it is evaluated against the other new knowledge that was not yet assimilated and is also present in the state of negotiation. If any new data emerges during evaluation, a deductive move back to the criteria is performed, and the cycle is repeated. It should also be noted that

---

142 Magnani, 2009: 35.

143 Ibid.: 35.

the hypothesis that is iteratively shaped on the left-hand side of the diagram is deductively verified by going back to the data which is expected from the criteria, while it receives the abductively produced knowledge that travels to it from the observed and through the explained. Such repetition interspersed with adjustments makes it possible to navigate the unsound logical reasoning that, by the nature of the process, has to deal with the defeasible incomplete information.[144]

The benefit of such an application of inference is that it permits the ignorance-preserving behaviour of the agents that are involved in problem-solving. The ignorance-preserving quality here points precisely to the fact that abduction is not a valid logical inference, and therefore, cannot produce dependable knowledge, but only lets creating the substitutes that bear enough relevance to suit creating the hypothesis. The disadvantage to the agents, as Chapter 4 explains in more detail, is connected to the agent's cognitive load, and is connected to the fact that abduction clusters the series of deductive and inductive moves, which can lead to stack overflow and cognitive shocks in agents in cases of extreme complexity spikes.

Composition with case studies

The present research used case-based work for field data collection as the best option in the situation of high uncertainty of the matters under investigation. Conventionally, case studies are well suited for the phenomena which unfold as the research takes place, within their real-life contexts, especially when the boundaries between the phenomena and their context are hard to define. Having an identifiable beginning and end, each case study affirms the changing environments qualitatively by constructing preliminary suggestions and tentative theories in parallel with collecting and evaluating the data. Case studies are not limited to observing, measuring and collating data for cross-case analysis. In my study, the cases were real-world scenarios that I, as a product lead and a day-to-day project manager, used to constructively intervene in the organisation's interactions in the problem space of production to maintain the continuity of communications and project deliveries. The application of the compositional method in case study research has made it possible to achieve sufficient parsimony of means, largely through the use of a repeatable process, discussed later in this thesis, as the production pipeline. The Appendix lists the most prominent cases I was involved in throughout my work at JX.

Besides the production pipeline, which applied to the whole team, I had specific protocols that I performed personally within each case. At the case initiation, I carried out the rapid knowledge acquisition phase to get as fast as possible to a stage where I could create the first dummy or a sketch of the

---

[144] Magnani, 2009: 16.

goals, requirements and methods of study. As soon as these were sufficiently clarified, I began the work on the case information architecture in parallel with the work in the field. I saw this phase as completed when the application of methods returned some first actionable knowledge. For example, in the case of selecting a software package for the film festival screenings,[145] the initial phase could be seen as passed when the data gathered via a combination of surveys and software vendor demos was substantial enough to be presented and evaluated across the organisation's departments. In Steve Maguire's terms, and as the rest of the section elucidates, the case-based work makes it possible to cut through the complexity of cases by reducing the operational data to the minimum which is required to make valid predictions accurate enough to be useful in the body of casework. In other words, filtering the knowledge which is useful for the epistemological grounding of the case, from which all the subsequent pertinent evaluations can be made.[146]

Such casing suggests that case studies need not be interpreted *nomothetically*, or in terms of general laws, to establish the causal models based on the accounts of associations between the variables. Instead, causality is understood as complex and contingent. This makes case study research appear, in the definition of sociologist David Byrne, as an *ideographic* project, taking each case as a unique instance to be understood on its own terms, and casing as an act which does not make peace with pre-given data sets or pre-constituted assumptions.[147] The specifically topological flavour of case studies describes software systems via establishing the spatially continuous entities, such as the variables, that are simultaneously present in the different facets of the complex domain under study.[148] For example, the constellations of people and technologies within the continuous planes of relations – the market and the organisation – become possible to grasp analytically via the variables such as the requirements and project stages. The continuities aid in finding a balanced approach to the problem of analysing the systemic complexity, where it needs to be sufficiently generalised to provide the grounds for the cross-case comparisons or drawing conclusions, while not being overly simplified to still be able to define all of the necessary conditions for the process of change.

The problem in developing the topological approach to case study research, as I learnt from the facts of my empirical study, is that the cases tend to be *elusive*, with the boundaries becoming blurred over time and requirements diminishing in their importance in the face of looming deadlines. This may explain the importance this study sees in spelling out the epistemology of cases, or how cases organise

---

145 See Appendix, CS1.

146 Maguire in Allen et al., 2011: 84 citing Moldoveanu, and Boisot and Child.

147 Byrne in Byrne and Ragin, 2009: 2.

148 Lury et al., 2012: 21.

what is known about the problem, in terms of the agent's expectations about what kinds of knowledge can be derived from the data and information available within a specific production context.[149] The ontological position of the present study is understood via Barad as discussed above, in relational terms of the agential realist framework.[150] The emphasis on epistemology throughout the thesis does not mean that the predominance of relations in defining the terms is here less important ontologically, but rather that thinking about the epistemology of the production method has played a greater role in my fieldwork. Since the ontological investigation is not the primary focus of my study, I generally align with Knorr-Cetina's outlook on the proliferation of local ontologies among the various levels of software abstraction, with the agents coming in a variety of forms, from standalone entities, such as experts or instruments, to firms, strategic alliances or whole industries.[151] As in the cognitive theorisation of Max Boisot, the agent can be present at any level as a 'system that receives, processes, and transmits data with sufficient intelligence to allow learning to take place,'[152] yet, as in Barad's agential realist account, the emphasis would still have to be on the relationality, rather than on what kind of agencies the agents contain. As Barad puts it, 'agency is an enactment, a matter of possibilities for reconfiguring entanglements.'[153]

Furthermore, epistemology in production is inextricably bound to the notion of audit fit for planning and review within extreme complexity situations. The abductive leap in developing a method suggests that relations and processes should be prioritised over thinking about objects. The latter only congeals as results and manifestations of such relations and processes, and thus can be largely used for retrospective evaluation of past strategic decisions. My agential realist stance assumes that even though software complexity is socially negotiated, the materials gleaned from the concrete cases cannot be discounted in favour of any particular theory. Technological complexity as a social phenomenon does not merely present a difficulty in understanding the software systems by the human operators, but also has a range of implications for the entire production lifecycle, including stakeholders and the system's technical parts, which makes it inextricably bound to both the process of generation of value in software capitalism and the society's regimes of governance. Knowledge thus is active, it decisively posits things in the real world. The method appears not only as something malleable enough to accommodate the study matter, but also as the assertion of action onto this matter, whereby using a verb, such as *casing*, to define a method of composing the cases as they get formalised and evolve within the situation of

---

149 Boisot and Canals in Boisot et al., 2007: 19.

150 Barad, 2011: 154.

151 Knorr-Cetina, 1999: 253.

152 Boisot in Boisot et al., 2007: 8.

153 Barad in Dolphijn and Tuin, 2012: 54.

knowledge labour, seems appropriate. Such a method, iteratively developed through the actions of capturing, queueing, composing or sorting, is mutually epistemic and ontic because the act of *doing the method* organises knowledge about the world together with the aspects of the world itself. The act of casing for my research is a way of rehearsing a responsibility in such *doing*, in the way that would bring a sense of rigour and accountability into the research practice.

From complexity to parsimony

During my fieldwork, I have adhered, as rigorously as the institutional ecology would allow, to the qualitative comparative analysis (QCA) case study framework for case studies developed by the sociologist Charles Ragin. QCA has been effective because it has allowed me to balance the knowledge derived from casework with the existing knowledge base and the change request fluctuations incoming from the stakeholder negotiations. QCA presupposes that it is necessary to gather in-depth insight into the variety of cases to capture the complexity of the context and to find, through gaining a better grasp of the problem space, the connections across cases. In the empirical case-based work, as it evolves in the present study, the approach includes three main steps: the initiation, the casework itself and the interpretation. Each stage represents an evolution in the processing of the case data, or, as the sociologists Benoît Rihoux and Bojana Lobe explain, something that QCA defines as the funnel of the complexity/parsimony continuum (Fig. 8).

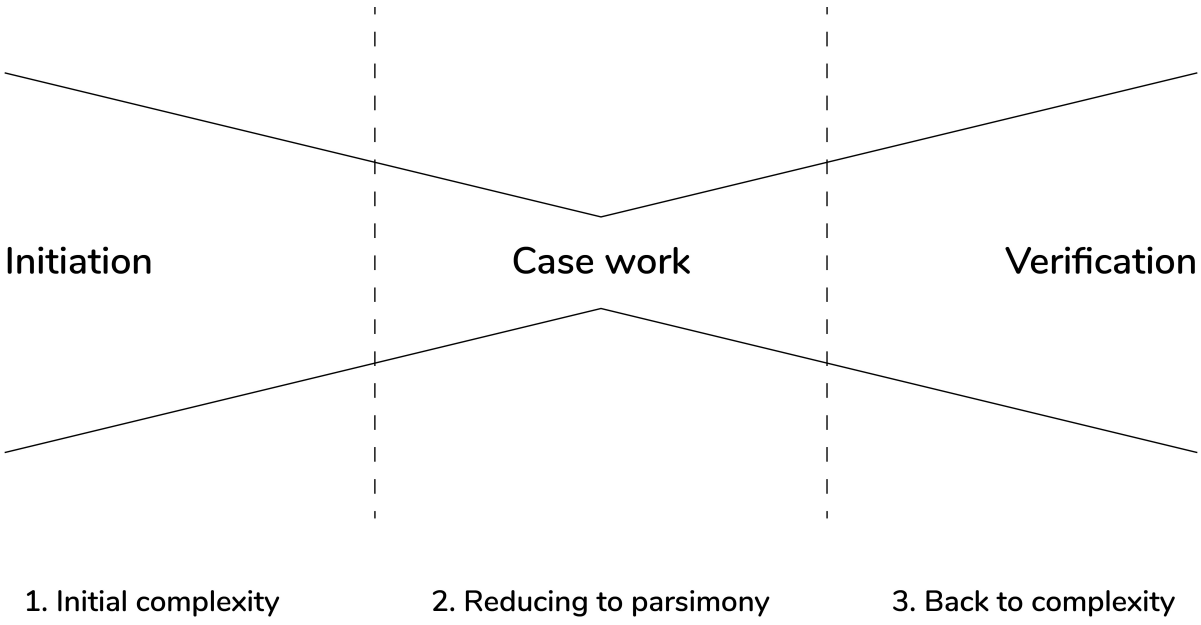

Fig. 8. The funnel of complexity in case study research. Adapted from: Rihoux and Lobe in Byrne and Ragin, 2009: 229.

Since the present study's challenge is to zoom into the issues pertaining to specifically technological systems and their production, the notion of complexity/parsimony continuum is used as a guide in situat-

ing empirical findings in *the problem space of production*, which makes the activity of casing inseparable from the construction of the epistemic infrastructure that concerns a specific technological system under investigation. At the beginning of each case, there is an overwhelming amount of reference materials, which are, however, irregular, not entirely accessible and therefore offer little value to understanding the problems. The goal at the case initiation is, therefore, to come up with a set of guidelines, usually in the form of a case report, that would propose a way to reduce the amount of knowledge and leave only the evidence specific to the case at hand. In the second stage, the complexity is further diminished through selecting the variables and coming up with the core formulation of the case problem, or any underlying principles. In the third stage, there is the reverse move back to the case, where the principle is verified through the application in the complex context of the field.[154]

The following example illustrates the use of the complexity funnel. In my empirical research, the case was initiated by a failure of the physical server, which led to the business stakeholders' concern that the failure might have meant the inaptness of the IT support contractors. In the first stage, the initial complexity had to be narrowed down by creating the initial draft of requirements and locating the variables in the form of specific actions, processes, technologies or people. Since the whole story of our collaboration with this IT contractor was not entirely familiar to me, I set out to draft a set of requirements based on the existing contract, my own knowledge of the system and the discussion with the system administrator. This resulted in a set of pain points that made us question whether the main issue was not with the IT contractors per se, but rather with the services they agreed to provide, and which we no longer required in the same capacity as years ago when the contract was written up. For example, nobody seemed to use that physical server since the time of the company's switch to remote operation.

In the second stage, we collected the data narrowing it down to key variables: the remote team's requirement of easy access to archives, parameters for the automatic backups of those archives and any service of the physical equipment remaining on site. In this stage, we have conducted group meetings and face-to-face interviews with the aim of generating enough material to be able to come up with recommendations for adjusting the requirements. As a result, we found out that we need to migrate the archives to the Google Workspace, splitting the access permissions between finance documents and editorial archives, and to decommission the old physical server. In the final phase of the case study, the interpretation, I sought to elucidate the causal connections between the cases and which of the variables give the most data in their comparison, or which of them may contain the explanatory value.

---

[154] Rihoux and Lobe in Byrne and Ragin, 2009: 230.

Thus, the move back to complexity was necessary and aided the understanding of the impact of change on the continuity – what makes the relationship persist, and how the continuous relation is re-established if the rupture occurs. Since the time when the server migration started, we have learnt that members of staff were not using the old server due to its low performance, and instead opted for the ad hoc remote storage solutions which were not secure and where no archival backup was in place. When the server was moved to the cloud, a series of onboarding workshops were planned to make sure that staff understood how to use the new cloud solution in a way that complies with security protocols. The new file system also meant that editorial, marketing and production teams had an opportunity to come together and agree on the folder hierarchy that would be convenient for all.

The act of casing

A case, in light of the compositional method, is present as an event of problem negotiation. It comes in the form of a collaboration between the agents performing different types of work and has to be auditable. The case can involve a variety of parties involved in casework – human or non-human agents, a member of the production team interacting with a user, a tester writing an automated test suite or an automated backup server communicating with the database. The base requirement, however, is that those involved in casework have a place to report to. Reporting may happen by internally produced means such as opening, closing and progress reports, presentations, spreadsheets, documents, design boards or development environments, as well as with the efforts of external audit such as assessment of key performance indicators (KPIs) or other performance reviews. In the ideal scenario, the case is tethered to the problem space of production via one or many support tickets, depending on the level of case complexity, and similar to tickets, multiple cases can be clustered, split or nested into one another. The modularity of cases makes them compatible with the bug-tracking software, and it should be generally observed that cases are, in fact, material enactments of tickets, and vice versa, that tickets are the means of symbolic manipulation of cases.

The activity of casing as the method of activation of the infrastructure on the one hand, and re-articulation of the problem space on the other enables to achieve both autonomy of the interdisciplinary research and accountability. This is possible because of the diffracted situatedness of the case, when the epistemology is continuously scrutinised from the position of the ontological awareness, presupposing the mutual influence between the researcher and the researched. Such processual awareness helps create a situation of the casing, where further categorisation, cross-case analysis and any other research procedures become possible. Casing, through becoming an integral part of the process of problem space composition, accounts for a spatialised and dynamic understanding of the field of knowledge,

and no fixed or geometric construction of the argument is either desired or necessary in the context where the ground of action itself is not stable, but drifting in response to that action.[155]

Speaking of the empirical application of the method, taking software products as technological systems makes the casing more specific in capturing the data and composing the case studies based on the formulation of problems as the structural approach to what is known about them. Technological systems have many interconnected parts across the variety of components that can be located in either the market or the organisation, and the disruptions can have echoes in many ways that would otherwise be hard to anticipate, explain or detect. By way of illustration, the problem I encountered in my fieldwork had been the back-end programming of the product's code. There are two sides to this problem. On the one hand, the product was limited in its functionality and did not have many third-party add-ons, meaning that we had to hire a contractor every time we needed a new feature. On the other hand, we did not require new features frequently enough to necessitate hiring a permanent member of staff who would, as the *toolsmith* in Brooks's surgical team, regularly create new code.

Thus, in the cases where existing plugins required maintenance or were not working as expected, we would suffer from the rupture in the relations between product and staff, as well as between staff members spanning across the entire space of operation of the technological system. For example, where the database query plugin returned, by error, the title of the journal post with unwanted non-alphabetic characters, the piece could not be shared on social media, which meant the stop press in the editorial schedule and disruption of work in the marketing department who could not initiate the campaign. The inability to solve the problematic situation directly by hiring the engineer for resolving each case individually, and the activity of casing the topology of the problem made it possible to accumulate enough evidence for hiring the back-end engineer after the fact, with a larger backlog of items.

Managing affect

When a project's model is created, the project managers are usually not required to use it for anything more than clarifying relations between the workload of staff assigned to do the tasks, the task durations or the budget allocated to them. This, however, means that a lesser consideration is given to the role of *affect* – the sphere of relations that condition bodily responses and emotions. While the detailed review of the wide and thoroughly researched body of affect scholarship is outside of the scope of the present thesis, some brief examination of its aspects pertaining to the production process is important. The interest of the present thesis here is to understand whether affect contributes to the mitigation of soft-

---

[155] Lury, 2021: 131.

ware complexity and, in case it does, in which way it manifests on the agenda of audit and other governance practices. This facet of my project points to a potentially fruitful study that could merit its own future trajectory of research, yet at present, some consideration needs to be given to the idea of a dual move that affective dimension may have in relation to the production process. This can be explained through the ideas borrowed from the philosopher Brian Massumi, the anthropologist Gregory Bateson, and the sociologist Patricia Clough, who have contributed to the theorisation of the aspects of affect relevant to this discussion.

Massumi approaches affect as a pre-individual capacity for the bodily response that precedes the visceral perception, which appears as the condition of linkage without considering the linked entities themselves.[156] Affect, Massumi observes, is indeterminate and therefore autonomous from whatever it inhabits – a 'pure holding-together (minus the held).'[157] Affect therefore precedes emotion in that it is present as a relation, or a condition for the possibility of emotion, which emerges in conscious perception as the narration of affect, implying that the affective force itself can never be fully exhausted within the sphere of emotions, and some excess always remains in the pre-individual realm. Bateson, theorising the collective moment of a behavioural unity, describes affect as the cultural dimension of a human group, which makes it form and inhabit a body of its behaviour as a coordinated whole. The unity of the collective behaviour, Bateson writes, 'is oriented towards affective satisfaction or dissatisfaction of the personalities'.[158] The cultural aspect of affect here implies that within a collectivity, the individual motives are grouped and balanced out to produce a consistent emotional flow which gets interwoven into the rest of the production activity. Clough, thinking through the politics of affect, discusses the post-disciplinary shift of governance to manipulation of affect with the aim of producing the subjects of value exchanges and control protocols. She comes from the definition of affect as the 'pre-individual bodily forces augmenting or diminishing a body's capacity to act',[159] which are open for engagement with the technologies, including both the technologies of circulation and distribution, as well as the technologies of control. In other words, despite its implicit and indeterminate character, affect is nevertheless susceptible to manipulation by market forces, which Clough summarises as 'capital accumulation in the domain of affect',[160] and to the organisational framing which de facto employs affect as a mediating mechanism to gain control over bodies.

---

156 Clough, 2018: 8 citing Massumi.

157 Massumi, 2002: 261.

158 Bateson, 1987: 75–76.

159 Clough, 2008: 1.

160 Ibid.: 2–3.

Establishing the notion of affect in this way makes it possible to think about its role in the mitigation of complexity as a concomitant dual operation of the individual and the group aspects of the affective dimension. As an autonomous category, affect makes it possible to evaluate the meaning and magnitude of software complexity resonance within the pre-individual, to a degree of affective relation's involvement with production. For example, it finds whether the effects of cognitive load during the extreme complexity spikes – the phenomena further explained in Chapter 5 – are in any way transformed by the existing sphere of organisational culture and to determine the complexity threshold after which the associated tacit knowledge is no longer effective. As a collective disposition, affect acts in its capacity for accumulating and transmitting embodied practices, and may help to explain why the tacit knowledge, which is necessary for creating the collective behaviour as a unity, does not stifle the audit efforts in complex production contexts. This links the affective dimension of production with the discussion of distributed collectivities, which Chapter 4 follows up on.

Because of such a twofold function, managing affect in professional DevOps is crucial, albeit not as easy to plan and review as other production practices, due to its being resistant to standard management methods. Such tools as a roadmap prove to be particularly effective for producing two-way transport between the individual and group affects. Due to its specific visual character, the roadmap provides the means both to discuss the existing tacit knowledges and to develop a collective behaviour as a unified body. In my fieldwork, I frequently turned to the roadmap, often in the form of a *Gantt chart*, which in practical terms allowed me to spatially distinguish the representation of the project's tasks from the time required to do them and the attribution to a member of staff carrying it out (Fig. 9).[161] Due to the inability to visualise the affective relations between the tasks, the Gantt instead appeared as the tool for the enactment of affect, serving as a support for real-life discussions in team meetings and the project's written correspondence. As the archival project case study discusses in the Appendix, the affective relation would unfold over time on the interpersonal level between the team members, which meant that the Gantt model could be regularly revisited, often to account for the reduced time in completion of tasks, alongside the increase in momentum.[162]

---

[161] For further discussion of my use of Gantt charts in the field, see Appendix, CS2.

[162] See Appendix, CS15.

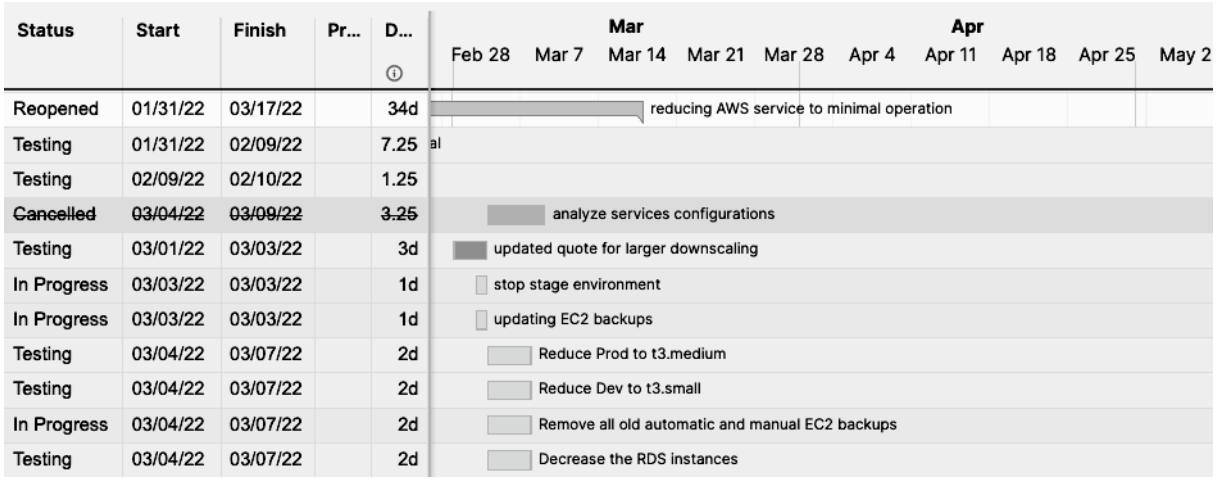

Fig. 9. The project Gantt.

The affective enactments around the Gantt made it possible to negotiate a common understanding of software complexity that the team has to deal with, establishing the possibility for the affective relation based on the fragments of the pre-individual forces which are always not-yet bearing the deterministic relationship to the collectivity, due to the complex and indeterminate nature of the production context. For example, sharing the attitudes in my fieldwork helped avoid the blockages in the communication channels created by misinterpretation, as the pre-individual traits were mutually explored and negotiated. To return briefly to the server migration field case mentioned earlier, during the process I was involved with many different activities, each being something that I've never dealt with before. As demonstrated partly in Fig. 9, the activities consisted of many moving parts which were occurring simultaneously and also involved different staff members and departments in and out of the organisation. The inaccessible files required contacting stakeholders for permissions and the IT department to activate those permissions; the destination cloud storage volumes required a new contract to acquire more storage space; the DevOps engineer was sending through the backup policies that had to be approved by the client operations department; the contract with the IT company had to be reviewed in the light of the changes to storage and backups; company staff had to be onboarded in terms of new access and security protocols, the map for the new file structure had to be drawn. During the complex coordination process, in parallel with the work communication itself, some affective relations with the team – for example, frustrations about the difficulty to communicate about all of the moving parts clearly and concisely – came up as equally important parts of the negotiation. I saw my role in this aspect of coordination to formalise the affective goal of the involved parties as understanding the relations of dependency and trust they shared regarding the system's complexity. Setting up such a goal, in hindsight, has helped maintain open minds throughout the team about thinking of how the project communication could be improved.

Methodologically, the affective dimension can be employed within the formal schema of QCA discussed earlier in the section, in its capacity as the outer limit for the sphere of emotional operations in the complexity-parsimony sequence. As an autonomous relation, it would allow identifying the relevant degree of parsimony that would allow coming up with a core formulation of the problem that a particular case study deals with. As a potential future trajectory of engaging with affect as a method of operations research, it can be critically approached as metric, or what Clough defines as *affect-itself.* Linking labour and affect within the context of the capitalist mode of production, Clough proposes a radical autonomy of affect through which it becomes an abstract category, alongside the abstraction of labour, as something that creates a possibility to measure the pre-individual capacities within the control mechanisms of production. To Clough, affect-itself is 'meant to address the becoming abstract, and therefore becoming subject to measure, of that which is seemingly disparate',[163] including the capital's involvement with the effects of software complexity for generating profits.

Chapter conclusion

In view of the overall goal of the present thesis to develop a model for the software production system lifecycle, this chapter described how such a model could be realised. Turning to the discussion of compositional methodology, I acknowledge the links that the matters of the present investigation have with the domain described by the ANT framework, specifically in terms of its multiplicity and fractality attributes. Turning to the core problem space dynamic which unfolds through the givens, goals and operators, I have emphasised that the operations are repeatable and tend to follow the patterns which make problem negotiations auditable. At the same time, the key trait of the epistemic infrastructure is its topology, which organises the existing evidence about the problems of production into a consistent body of knowledge. The second theme of the chapter has discussed some of the aspects of employing diffraction and abduction methods in situations of high uncertainty. Diffraction was seen as the necessary aspect for problem composition every step of the way, as the act of cutting things apart before putting them together, but also the putting-together being an iterative activity of continuous overlaying of shapes in search of the effects of difference, up to the point when cutting together-apart becomes one and the same event. Simultaneously, the method of abductive modelling deals with such tentative encounters to create hypotheses that enable the preservation of the unknown without employing any assumptions disconnected from the facts established in the field.

Lastly, I present the fieldwork as the crucible where the composition, diffraction and abduction are brought together through their consistent utilisation in case studies. Through casing, the materiality of

---

[163] Clough, 2018: 3.

organisation and market relations as established in Chapter 1 become entangled with the performativity and affect of real-world production situations. The main question that this chapter poses is, what kind of methodology is required to study the complexity effects in the production system of software capitalism? In view that the encounters between the software production system and the organisational entities are fractal, the answer is that the study is possible by creating a template for abductive modelling applicable to the various abstraction levels, which would elucidate the relationships in a scale-free way. Creating a template would facilitate creating the models in a variety of production scenarios, being able to navigate the uncertainty by preserving ignorance where the data is unavailable.

This concludes the literature review and the methodology chapters of the present thesis, and therefore it is possible to sum up what the study saw thus far, and what are its next steps. Centralised technological systems have been argued in Chapter 1 as characterised by their rigid internal hierarchy of control. Systems of this kind are appropriate for production scenarios of low or limited complexity, such as the factory production of physical objects. Going forward, the research is interested in looking into the systems that I refer to as distributed – or those which are capable of unlimited increases in complexity, such as most software systems, and therefore cannot be managed in a centralised manner. The focus on distributed systems, I contend, is urgent due to the ubiquitous utilisation of software systems in production. Admittedly, production systems used in mass manufacturing in most contemporary factories already do not qualify as simple systems for this argument. They are no longer limited from the moment they start using software in their processes because the software is capable of abstraction layering, and thus of infinite complexity, even if the physical quantities of materials on the shop floor are limited by the actual size of the shop.

In the following three chapters, I think of the ways of understanding the governance of distributed systems by creating a more nuanced production design lifecycle model (Fig. 4, at the beginning of this chapter). Chapter 3 focuses on the left and top of the diagram. It discusses the notion of epistemic infrastructure that contains definitions of the system components, and the process of deployment of software, which makes software available to its users. Chapter 4 discusses the part of the production process which is crucial for understanding the involvement of governance with software complexity – the practice of audit. This practice is understood as *control of control*, in alignment with the accountancy theory of Michael Power,[164] and is prominently present in the negotiation of meanings that happens in the problem space of production, located in the right part of the lifecycle diagram. Chapter 5 moves to the involvement of governance with computation and cognition through the process of integration of

---

[164] Power, 1999: 12.

knowledge, found at the diagram's bottom. The integration activity concludes the description of the research model and enables me to apply the research findings to the wider context of software and cultural studies. This is possible due to the effects complexity bears on computation, a process which is present as a common trait of much of the research carried out in these fields.

Chapter 3. The epistemic infrastructure as code

As the previous two chapters argue, there is a decisive split within any knowledge-based production into the new and already confirmed knowledge. This chapter is interested in the epistemological constituents of the system, the infrastructure of existing evidence, and the process of deployment of the system based on this evidence, practised in DevOps as Continuous Delivery and argued in this thesis as the topological machine. In this sense, I treat software production as the production of means of production, and simultaneously as the reproduction of the labour ecology of the entire business value stream. This point of view facilitates the establishment of a notion of *epistemic infrastructure as code* (EIAC), a notion inspired by the DevOps concept of *infrastructure as code*, to be able to account for the complexity that arises in the relations between software components in Continuous Delivery.

The three sections of this chapter look at the three research agendas common to most complex deployments. Starting from the deployment pipeline, the first section explains it as a core operations pattern used to treat the complexity in component-based systems. It does this by first establishing more firmly a difference between industrial mass manufacturing and the production of software, and then discussing the delivery pipeline and its components as they are used in Continuous Delivery. The second section turns to the topology of production system design through its two popular varieties: functional, which is used in centrally-managed systems, and stream-aligned, a widely accepted standard for complex non-hierarchical production situations. Section three focuses on the specificity of the circulation of knowledge through its three main features, the structural coincidence between the source code and the organisation's structure, the conceptual fuzziness of software as a product and software as a process, and the disintegration tendency in technical systems.

Deployment pipeline as the topological machine

This section brings together the two notions, of the deployment pipeline and the topological machine, to be able to work towards a Continuous Delivery critique. This is done here through three themes. I begin by working through some specific assumptions that may be risky if carried over to the sphere of software production from mass manufacturing. Next, I explain the DevOps notion of deployment pipeline and how it fits within the general schema of the production system proposed in the present study. Lastly, I describe the central topological configurations of production teams that would allow understanding in which way the continuous deployments are made possible within the overall organisa-

tional structure of software capitalism. Throughout the section, I assume that the central feature of the topological Continuous Delivery production method is its continuity, which means that it describes the space in terms of continuous surfaces between the points in space, separately from their metric attributes. In this type of deployment, the distances between the points can be flexible and relative. For example, while the straight line can appear as the shortest distance between the two points on the geometrical map, it does not account for hills and valleys that appear on the way from one point to another. The deployment pipeline as the topological machine, explored in this section, activates the algorithms for creating the points in the problem space of production, along with the continuous relations that bridge the gap between the points, creating the production system's topology of negotiations. The increase in complexity in the production of topologies, however, is unavoidable because the topological promise of continuity is not necessarily kept by the machinic deployment, which works through its breakdowns, rather than by warranting the delivery of a required outcome.

Industry antipatterns

In the world of manufacturing physical objects, tasks are repetitive, activities are reasonably predictable, and the resources should be located in one place at a time. In software product development, many tasks are unique, project requirements constantly change, and the information-based output can reside in multiple places simultaneously. The several potentially risky antipatterns emphasised by the operations research of Stefan Thomke and Donald Reinertsen, often cited in the professional DevOps literature, are high utilisation, large batches, forward planning and adding new features.

High utilisation. First, speaking of exorbitant resource use antipattern, high utilisation in software production adds complexity because variable tasks form long queues and have longer waiting times. Such observation may seem to come into contradiction with the Marxist critique, which postulates that the extracted value will continue to increase alongside the increase in the utilisation of factory machines. This explains why early capitalist production favoured day and night shifts work arrangements that would allow running the machines at their full capacity. The principle of full utilisation, however, fails to apply in workflows where the tasks are more variable than on the factory's assembly line. While in repetition-based workflows an increase in work tends to increase execution amount in the same proportion, in qualitative workflows such as software production, the outcome of the increase in workload is uncertain. One of the outcomes is the miscalculation in completion time, where the teams are overcommitted most of the time, which is dealt with in Agile through the practices of story point estimation and tracking the team velocity. Another facet of the same problem is the accumulation of tasks in the backlog, which is specifically addressed by the dedicated team member who can mitigate the uncertain time of addressing backlog tasks by negotiating the release cadence with the stakeholders. For

example, if the tasks added to the backlog queue deviate too far from the initial release objectives, it might be possible to treat them as change requests and transfer them to a different release where the objectives might be amended to address them.

The issue seems to stem from the fact that the inventory – that is, things produced or otherwise contained in the space of production – are not immediately visible. Whereas in industrial manufacturing the accumulation of stock can be noticed in storage facilities, the informational output of software production efforts has no physical signs and is only expressed in the documentation, test procedures and results, or infrastructure code instructions. The serious excesses in resource rent or data accumulation can go unnoticed for months or years, even in small organisations – for example, in my fieldwork the AWS infrastructure was running on a configuration which was at least three times the size of what was required, and had been generating regular automatic backups that were never used and never erased, which meant increasing expenses which could not be diagnosed and addressed without the involvement of an experienced system administrator. This uncertainty had added to my product lead duties in the field, meaning that even if I knew about the excess, I could not address the issue without creating a convincing case that would allow the stakeholders to evaluate if the expense of hiring a dedicated contractor to solve the problem would make more business sense than leaving the problem unaddressed and continue paying for the unused resources. In the context of the present argument, it should be noted that component-based systems, as opposed to monolith software blocks that have no internal divisions, are more resilient to the high utilisation issue. One of the ways to balance the load in component-based systems is to abstract the issues which threaten to cause significant delays as new components. This means that issues of potentially variable complexity can be addressed in a more granular scaling fashion than in the monolithic system.

Furthermore, the high utilisation has an additional negative side effect, in which the management assumes that the sooner the project is started, the sooner it will be finished. With this consideration in mind, the management proceeds to exploit any staff downtime by starting new projects earlier to fill in the gaps. However, this scenario means that the new work will be undertaken slowly and will be constantly interrupted to finish off the tasks related to other work in progress. This is as risky as doing slow and intermittent work under any other circumstances. The risk is caused by the fact that the software production outputs are highly perishable and can become obsolete before any version worthy of release can be achieved. The warnings that real-life situations present is that often the work still has to be undertaken to grab the opportunity of a resource that will not be available later, or when there is at least some assurance that the work done is not going to perish soon. In my fieldwork, we have undertaken overlapping tasks with relative success, albeit causing some unnecessary frustration in the team. One of the product features we had to deliver was city and country landing pages for the travel section

of the organisation's website. This required a back-end engineer, who had a bit of time available, while the rest of the team was busy on another, more urgent, release. It was also known that if we don't seize the opportunity to use the back-end at this specific moment, they will be switched to a different project later, and the decision was made to initiate the work. While it has allowed the team to deliver the city and country landing pages feature on time, the back-end was not able to communicate effectively to the front-end staff who were busy on other projects, which caused additional stress, highlighted in the project retrospective meeting.

Large batches. The second potentially dangerous industry assumption is large batches. Feeding the large new pieces of work into the main production adds complexity because larger bits of new material are harder to integrate and test, especially when the new pieces have been developed separately from the main line and are not easily compatible. This antipattern sees the production teams waiting until releasing a larger batch of work, rather than delivering continuously. Reducing the batch sizes is crucial for lean manufacturing principle, and works through optimisation of the two primary costs: the transaction and the holding cost. As batch sizes become larger, average inventory levels rise, which raises holding costs. But at the same time, transaction costs decrease because it takes fewer transactions to service demand. With these two parameters in mind, the optimal batch size would be located in the area where the combined holding and transaction costs are the lowest. By making frequent releases its essential requirement, hence its name, Continuous Delivery claims to decrease production expenditure in two ways. One is the cost of risk updating the parts of code which may perish while waiting for the release, another is cutting the cost of integration. The rationale here is that any testing is easier to carry out if the deployment is done on the same day, as it supports the easier localisation and troubleshooting of any new changes.

Forward planning. The third complexity-adding antipattern is excessive forward planning. In software production, creating a detailed plan at the beginning of the project is not optimal because means and ends change during production. Insisting on carrying out the work as planned despite changing circumstances may also lead to increased complexity in relation to legacy code. The usual rationale for forward planning is that if the production adheres to the original development plan as closely as possible, this will help to deliver the required features on time. In a real-life context, however, it proves to be ineffective because new knowledge is being generated throughout all moments of the software production process, creating conditions which were not available for upfront planning. Furthermore, the frameworks and tools used evolve alongside the product itself. Thus, on the project level, as the Agile method prescribes, the tactics have to be continuously adjusted. As Chapter 1 explains, the stage-gate or waterfall principles can still be beneficial for high-level strategic planning.

There are good reasons why a related trend, which assumes that all the requirements can be met in the first release, is also discouraged in Continuous Delivery. For one, it makes teams prefer less risky solutions to avoid any errors in the first release, which usually means underreporting of errors and an increase in momentum at the expense of the overall resilience of the production system. Another reason is that overly safe releases often result in delivering less value in the value stream, leaving the users wondering how the proposed product is better than any competitor products that might already be available on the market. Finally, this aggravates all the negative effects of the stage-gate process, such as decreased throughput and late discovery of issues, when it's more expensive to solve them. The practice of frequent deployments in Continuous Delivery is generally considered a way of assuring that an environment remains safe from failures and experiments. This is achieved either by deploying in multiple testing environments or by being able to rapidly roll back in case of an error. However, Thomke and Reinertsen admit that creating an environment open to failure is not easy: failure is negated in many organisational cultures, for example, those known as 'zero tolerance for failure' or Six Sigma, and managers who fail may be putting their careers at serious risk.[165]

Another issue frequently referred to in relation to this problem is the number of hand-offs between teams, which adds more time added to the task than is accounted for. As Gene Kim notes, assuming that each code change would require the full cycle of hand-offs between network, server, database and other teams, factoring in the time required for testing and approvals the task will cause an exponential increase to the completion timeline.[166] This is addressed, as the team topology section in this chapter finds, by decreasing the number of hand-offs by orienting teams along the streams rather than functionally. The problem still exists, though, as it can only be addressed partially by this method.

Adding new features. The fourth antipattern that needs to be mentioned is adding features through change requests. While there's the risk of cementing the plan from the very outset, this case presents the opposite extreme: too much deviation from the initial strategy by adding new features throughout the production process, which may be equally risky. Here, the complexity may increase because the incoming requirements for new features may obscure the larger task, which is a thorough definition of the main problem. The rationale for this antipattern is that the more features the product has, the more business value there is. While it is clear that focusing on fewer features will make delivering the requirements easier, it is not always easy to keep the product simple due to the two interdependent factors.

---

[165] Thomke and Reinertsen, 2012.

[166] Kim et al., 2016: 435.

The first factor is that extra effort is required to define the problem. This stage is frequently overlooked since the task of understanding the real underlying problem often looks easier than it is. The second issue connected to it is that to define the problem, the team has to spend time going through the stage of setting the goals and then proceed through the multiple rounds of testing and experimenting to understand if the strategy addresses the problem, and, importantly if the problem addressed is in sync with the business value stream in terms of customer delivery. The initial stage is sometimes referred to as the discovery stage and is worth the additional investment since it then contributes to significantly limiting the production efforts to the features critical to the value of the specific product. Mitigation usually proposed in Agile-inspired methodologies is setting up an explicit condition, referred to as the *definition of done*. The definition usually comes in the form of a policy that describes the criteria which a ticket or other piece of work has to meet after which no more work is required – for example, passing specific tests or gaining approvals from certain parts of organisations. Regarding the antipattern that displaces design efforts with marketing, organisational culture needs to shift towards refactoring and the unified technical treatment of features, which would be capable of catering to the diverse marketing requirements.

The problematic assumptions outlined above can be seen as the reasons why Continuous Delivery addresses the complexity in software systems through component deployments. Splitting the system up into components, as this chapter finds later, promotes sufficient flexibility in both parts of the system that require the work and the team topologies that carry out the work. Complexity, however, is not ignored but is made more accessible for analysis as it is abstracted into a matter of compatibility between the components, something which can be addressed strategically by operations in the deployment stage, rather than in an intermittent manner throughout the writing of code, testing and other production activities.

Component types and rationale

This section works towards a more thorough understanding of the epistemic infrastructure method in its application to the study of a software system. To do this, it is necessary to turn to the notion of a component, which in Continuous Delivery serves as the primary means of mitigating the risks associated with deployments and is a key category in negotiating meanings in the problem space of production. It is appropriate in this context to turn to the notion of a component in David Farley's description of the Continuous Delivery framework. Here it is defined as a conceptual device that adds a new layer of abstraction to the deployment process so that the complexity of compatibility between the different parts of the software application can be dealt with separately from code creation, testing and other

production activities.[167] A component, Farley suggests, is 'a reasonably large-scale code structure within an application, with a well-defined API, that could potentially be swapped out for another implementation.'[168] It should be emphasised that in this sense the component can be deployed independently into the working application without affecting its general performance and therefore bears a coherent set of behaviours.[169]

A component-based application is viewed in this context as an application where the code base is split into a number of discrete parts which relate to one another in a stable and well-defined way. Each logical part of the software system exists separately from another, which makes such a construction different from the alternative, which is a monolith that has no segmentation between the parts and thus contains all of the complexity inside one system, which makes it harder to understand. The component-based approaches to system deployments are argued as more efficient in at least four ways. First, they allow for the discrete analysis, since the components divide the problem space into a series of self-contained areas. Second, they have different lifecycles, which makes it possible to analyse different parts of the system on different temporal scales. Third, they allow dividing the responsibilities of teams, making it easier to adhere their production schedules to the practices of audit. Lastly, and most importantly, they abstract the complexity that underpins the functioning of components, creating a new relationship between the inside and the outside of the components. The relation is manifest in the creation and maintenance of boundaries, a process which is, as Barad demonstrates, indispensable for making meanings and is present as the instances of power that have material consequences.[170] On the inside, this gives the teams the freedom to optimise, test and maintain each component separately. On the outside, it creates a possibility for symbolic manipulation of components which serves as an interface with the internal organisation's governance protocols and can be effectively reported on and accessed by the executive staff. Simultaneously, the main problem is that due to their differences, components introduce new uncertainties in the deployment process.[171]

As far as Continuous Delivery is concerned, the software system consists of four main components: *data, host environment, configuration* and *executable code*. To be compliant with delivery requirements, all components, save for the data, are required to be written down as code to be repeatable and ready for deployment from the version control system. Without a doubt, *data* take the shape of code too, yet it

---

[167] Humble and Farley, 2010: 345.

[168] Ibid.: 345.

[169] Ibid.: 356.

[170] Barad, 1996: 182.

[171] Humble and Farley, 2010: 356.

would be more precise to refer to them as a collection of basic units of meaning made accessible for further interpretation. An *environment* is a complete set of resources that the software system as a whole needs to operate.[172] This consists of hardware, such as processing units (CPU) and memory on the one hand, and an operating system and middleware, such as the application, database and web servers, and messaging systems on the other. *Configuration*, put simply, is the desired state of any part of the system. What makes it different from configuration uses in other production paradigms is that in Continuous Delivery, the configuration is managed through version control, together with source code and documentation, thus making it deployable via the pipeline as any other part of the infrastructure. *Executable code* is the code binary which is compiled from the source code created by the developers, contained in the appropriate production branch and delivered to the environment. It is built every time the source code is changed, and goes through a series of automated test procedures, or a test suite, which is also ideally included in the pipeline.

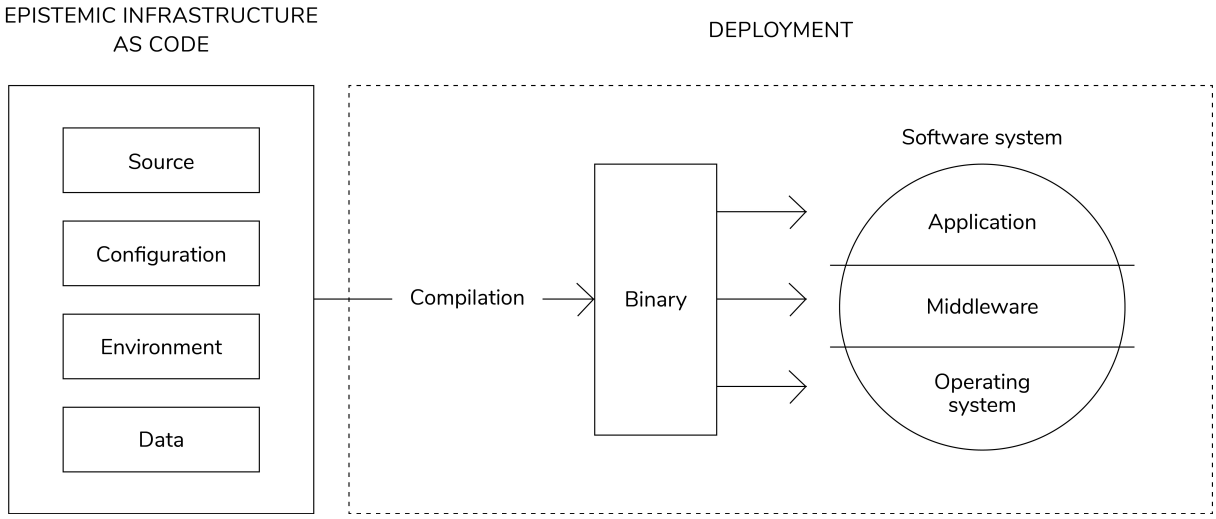

Fig. 10. The layered delivery of the software system during deployment.

Deployment, which is a set of processes, described in the infrastructure code and performed by a computer, makes the software system available to users through various sorts of engagements with all four system's components. As demonstrated in Fig. 10, the *compilation* is one of the important steps in deployment, during which a computer translates the information derived from EIAC to produce the executable binary. The binary is then consecutively delivered to production in layers. This comes from the DevOps best practice principle that the safest deployments are easier to achieve if they are carried out with settings which are known to be performing well and error-free. Layered deployment makes this principle easy to follow, since if there are any errors in deployment of the preceding layer, there is

[172] Humble and Farley, 2010: 277.

no deployment of the consecutive layers. The base layer is the hardware configuration – usually an operating system. The second layer is middleware, which includes all the auxiliary software that the main system depends on. The two layers are deployed as separate layers to make sure they are configured and running well before rolling out the top layer, which is the system, its own configuration and any associated apps and services.[173]

Technical and organisational facets of the deployment pipeline

According to the Continuous Delivery approach, any production risks can be avoided by releasing as frequently as possible – hence the delivery is *continuous*. A pattern central to Continuous Delivery is the *deployment pipeline,* or the automated implementation of a software system's build, deploy, test and release processes. The goal of reducing risks in releases deals with software complexity in the sense defined for this study, particularly when maintaining the account of the complex relations between the components in software system deployments. For mitigating the complexity effects, Continuous Delivery utilises the deployment pipeline in two ways. On the one hand, it serves as the abstraction layer that supports the focus on complexity in relations between the application's components without going into the details of development work. On the other hand, it is tightly linked and evolves together with the company's technology value stream.

Where the latter utilisation engages the stakeholders in understanding how technology converts strategy into value delivered to customers, the former creates a blueprint for the technological solution that makes the delivery happen. Importantly, the deployment pipeline does not merely compile the application source code, but also activates a complete specification of what resources are to be used, builds the infrastructure and initiates all the testing stages the code has to go through. The specification for the pipeline is written down in the code, which is referred to in DevOps, appropriately, as *infrastructure as code*. This makes the deployment pipeline appear more than a means of production, but rather a tool that combines the process of creating the means of production, that is, the infrastructure, with the outcomes of production – the executable binary.

Automation and repeatability of the deployment pipeline are relevant for the present software complexity argument in two respects. On the one hand, it makes the deployment process auditable, which is important in cases of malfunction, since the feedback comes immediately. On the other hand, once the production of the problem space of production as a whole is described in code, it means that it can be reproduced any number of times, for example as a test copy for troubleshooting. In fact, in DevOps,

---

[173] Humble and Farley, 2010: 162.

deployment is only acceptable when it is automated because it is mainly through the verification of the automation scripts that the deployment's policy compliance is established. If any of the deployment stages were done manually, it prohibits risk assessment and is thus considered a breach.[174] Furthermore, having infrastructure written down as code makes it possible to carry out a variety of administrative procedures to it, such as change management to understand who made alterations to environments and why, assigning team members responsible for approving the deployments and managing access permissions, or requiring the updated version of the documentation to complement the releases. In other words, an organisation gains access to scalable control of its knowledge frameworks. Since the code can be used algorithmically to recreate the epistemic infrastructure from scratch, there exists an assurance that, regardless of the number of times it's been created, all of its properties are identical, down to the very minute details. It also reduces the cost of errors, since the epistemic infrastructure can be tested and debugged as any other code, rolled back to a previous version and streamed continuously – or in terms of topological framework, provided as a service.

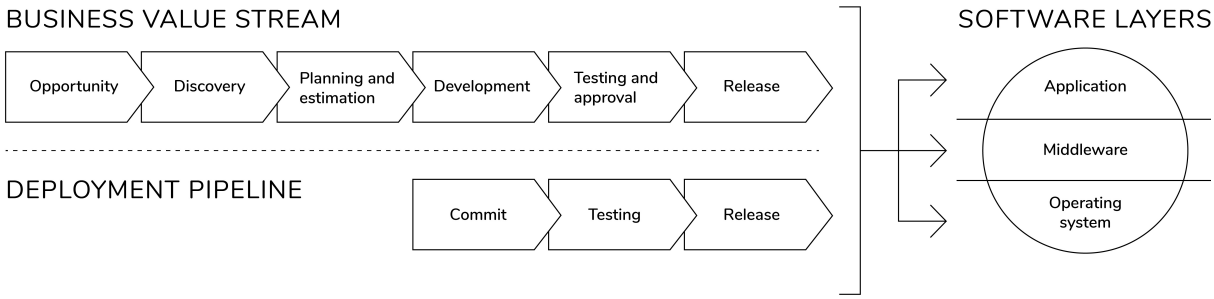

Fig. 11. The layered software product construction.

The construction of the deployment pipeline tends to closely adhere to the value stream of the organisation's business model (Fig. 11). Since no organisation is like the other, the deployment pipeline will also vary in each particular case. The general pattern that each pipeline follows is that it always begins with the Commit stage, contains some Test stages, and ends with a Release stage. The Commit stage would consist of creating the source code and its initial analysis. The Test stage deals with automated tests and depends on how sophisticated the organisation's test suites are. In the Release stage, the pipeline makes the code available for release into any of the organisation's environments. This stage depends on the company's deployment policy and team topology. Any releases would initially be deployed into an environment only accessible to a limited number of users, such as stakeholders, who would have to accept it before the release is deployed into the publicly available production environment. It is important to pay attention to the differences between the technical side of deployment in

[174] Humble and Farley, 2010: 438.

the previous, Fig. 10, and the deployment stages in Fig. 11. It may become clear that while the underlying mechanic is relatively invariant and always follows a pattern of compiling the code and deploying it, in the context of organisational structure the deployment process can be very different depending on the organisation's structure and the procedures of approvals or compliance.

There need not be any ambiguity about which software is at stake when investigating software complexity as the property of the problem space of production. This becomes clear once production is presented as a layered activity: while there can be no doubt that all layers that Continuous Delivery deals with are the *software* in the broad sense, it is specifically when the whole stack is produced that the complexity of the constituent interrelations is the most relevant for the present study. Furthermore, complexity thus understood can be investigated on a much broader scale, since the tensions between the system's components, as the previous section demonstrated, can be traced from the topological orientation of software to the team topology. There is thus a continuity that links the configuration of the organisation's departments, its development operations, delivery pipeline, technology value stream and the software system as it is presented to the user.

Deployment of the topological machine

This section has found that a deployment pipeline's role is to activate the business value stream and regulate the deployment activities. The pipeline deals neither with the database design nor with the code contained in the repositories of version control systems. Instead, it operates on a higher level of abstraction and instructs the decisions on which databases and which versions of the system are deployed in the specific release, how to compile the application code and which test suite to run. As a mechanism of governance, it is an example of a *topological machine,* a term used in software studies to refer to the automatic production of space that encapsulates specific behaviours and conditions users through its internal structure, for example, when analysing the database design. The two notions, the topological and the machinic, that define a *topological machine* give it a very specific meaning that sheds light on what Continuous Delivery does. On the one hand, the delivery is topological, meaning that the stakeholders are dealing with a specific spatial configuration, based on the relations between points or nodes. On the other hand, the *machine* connotation implies the computational core of the spatial configuration, based on the event sequencing using *if-then* boolean logic.

The purpose of defining the deployment pipeline as a topological machine in the present argument is to be able to explain the continuity of the deployment dysfunction, which appears as a defining principle throughout the production process. The dysfunctionality can be understood on two levels. The primary cause of the break is that the operation of the software system and the operation of the

pipeline are two different processes that operate independently. As the next section finds, the stream-aligned team topology principle addresses some of the frictions that arise from the traditional divide between development and operations, yet there remains the essential antagonism between the parts which was pointed out by some of the pioneering thinkers of automation in computer science. For example, the mathematician John von Neumann, who addressed the matter in his theory of self-reproducing automata, notes that within any unity some parts act antagonistically to other parts. As cited by the historian Philip Mirowski, von Neumann writes: 'it has already happened in the introduction of mass production into industry that you are no longer producing the product, but you are producing something which will produce the product … the relationship is getting looser.'[175] The difference in planning and production between producing the product and producing something that produces the product means that the primary automaton functions in parallel, with its parts running simultaneously on different features, there is a possibility of conflict. In other words, while gaining the production power in automated deployments, some of this gain has to be spent in addressing the complications that inevitably arise in the system which is functioning separately from the functioning of the production system.

In the second instance, the dysfunctionality can be accounted for through the optics of French philosophers Deleuze and Guattari, who contend that the *plane of immanence*, or the internal operations of capital, upon which the components and relations of production are organised, has dysfunction as a necessary precondition of production.[176] This is the case because of the increasing involvement of governance with operations, and the concomitant growing concern with the management of dependencies, which is only possible if the gaps and breakages are present and made available. While this problem will be more fully discussed in the next chapter, it has to be mentioned here that dysfunction is, in fact, essential to the computational sequencing of deployments, where the instructions are carried out step by step, and the space does not have a reliable, pre-determined quality, instead being produced contingently based on the *if-then* outcome of each consecutive instruction. Thinking with Melvin Conway, the disruption is therefore required by the organisation that structures itself to coincide with the system that it aims to repair, and also for the system's audit, since any effective planning and review of software is only possible where the communications of organisation and the software construction coincide.[177] Once the dysfunction is understood at the operative principle of Continuous Delivery, it

---

[175] Mirowski, 2002: 149 citing von Neumann.

[176] Deleuze and Guattari, 1983: 151.

[177] Conway, 1968: 28.

becomes possible to organise the disruption-oriented workflow, for which Agile methodology and its tools, such as the support tickets, are essential.

It is thus a topological machine that does not only define the continuities, but also creates the conditions of the possibilities for such continuities to emerge. The machine quality of the deployment pipeline lies in the fact that the pipeline itself is nothing more than the executable code which, being itself open to governance, unfolds spatial configurations fit for the audit. In fact, the best DevOps practice demands the auditability to be ingrained so deeply in the deployment that it can be considered an architectural property. The key to auditability is to implement any change via EIAC: 'there is no better audit trail than a record of exactly which change was made to production, when, and who authorised it. The deployment pipeline provides exactly such a facility.'[178] The code of a topological machine in this context is an ultimate account of the business value stream, and is referred to as an *infrastructure as code*, precisely because it is capable of such functions as starting up and configuring new resources and processes. In the present chapter, the thesis aims to gather enough evidence to be able to expand the DevOps notion of the infrastructure as code into the realm of continuously delivered knowledge about the software system as EIAC – the knowledge infrastructure which is provided as a service and which evolves alongside the problem space of production.

The team topology principle

This section examines two of the most prominent topological team arrangements: functional and stream-aligned. Among the many organisation design approaches, these are selected for two reasons. On the one hand, this is because the rest of the more intricate approaches still would have either one of the two topologies at its core. On the other hand, because the functional approach is more traditional, looking at it in comparison to the stream alignment paradigm makes it possible to understand what kind of team arrangement is required for the effective implementation of the Continuous Delivery method. The discussion has to begin from understanding that a team in any such arrangement is taken, in Matthew Skelton's terms, as the smallest entity of delivery within the organisation. For analytical purposes, the team is usually defined as a stable group of five to nine people who collaborate toward a shared goal.[179] The team comes with the interface, which can be referred to as an API, or an application programming interface. This is the same term as the one used in source code programming, and its use for teams is justified because it serves the purpose of succinctly describing the attrib-

---

[178] Humble and Farley, 2010: 273.

[179] Skelton and Pais, 2019: Ch.3.

utes of the team's relations to other components of the problem space of production. Such attributes are *code* produced by the team, *configuration*, technical *documentation*, *culture*, *communication* and *roadmap*.[180]

Each of these attributes describes how easy it is to integrate the team into the technological system. *Source code* is what the team produces in terms of tangible outcomes: libraries, user interfaces, components or other assets. Team *configuration* management describes how the team treats the changes. It is often synonymous with version control, which is a mechanism for keeping multiple versions of data so that when the change is made to any file, it is still possible to access the previous revisions. *Documentation* is frequently thought of as an audit technique that makes it possible to quickly find information about a specific aspect of work. This is required, as Farley notes, either to inform a team about something new, to refresh the memory, or in the case of troubleshooting, to be able to locate the relevant changes that caused the problem.[181] The caveat is that, of course, the fact that something is written in the documentation does not guarantee that the event has taken place,[182] and in this sense, EIAC provides a more reliable audit trail – the 'automation over documentation' DevOps principle.[183] *Culture* is present, as defined in Chapter 2, as the unity of shared meanings organised to promote cooperation. *Communication* describes the practical approaches, in terms of which tools and which types of information are being circulated: chat messaging, comments in version control systems and meeting cadences. The *roadmap* accounts for the way in which the team sequences its work: how is the current plan structured, what is the velocity and how the tasks are prioritised.

The key concern of the topological approach to designing the teams comes from the concern about the *cognitive load* of the respective team members. Cognitive load, which I discuss in more detail in Chapter 5 in relation to the integration process, can be understood at this moment, via psychologist John Sweller, who coined the term to refer to the 'total amount of mental effort being used in the working memory,'[184] in other words, the capacity for retaining and processing information either by individual workers or teams. The concern here is that the more complex the system that an organisation builds, the higher the cognitive demands on the teams' efforts. The topological approach works by splitting the teams to address the problem space in a way that facilitates managing the cognitive load. An intuitive way of estimating the distribution of cognitive load is to think about the organisational

---

[180] Skelton and Pais, 2019: Ch.3.

[181] Humble and Farley, 2010: 280.

[182] Ibid.: 437.

[183] Ibid.: 437.

[184] Sweller, 1988: 257–285.

body in terms of its communication lines that naturally divide it into something DevOps refers to as *fracture planes.*

The idea of such planes comes from the early notion of the large single-block software systems which are sometimes referred to as monoliths – or large stones – which are hard to understand when approached in their wholeness, but lend themselves to more productive analytical engagements when split, similar to large stones, alongside their natural seams. The natural seams, or fracture planes, help locate the key communication channels, which appear in accordance with the structural coincidence principle known as Conway's Law. To Conway, 'organisations which design systems … are constrained to produce designs which are copies of the communication structures of these organisations.'[185] This means that when the teams are formed according to the fracture planes, there are no unnecessary dependencies or extra communication efforts between them. A metaphorical interpretation of the problem space of production in terms of fracture planes is useful when designing the team topology, which needs to keep the software boundaries aligned with the various parts of the business domain. Fracture planes can be either of a more functional kind, such as change cadence, risk or regulatory compliance, or based on specific challenges, such as a technological solution to be implemented, improving the performance of a specific part of the system, or dealing with a particular user persona.

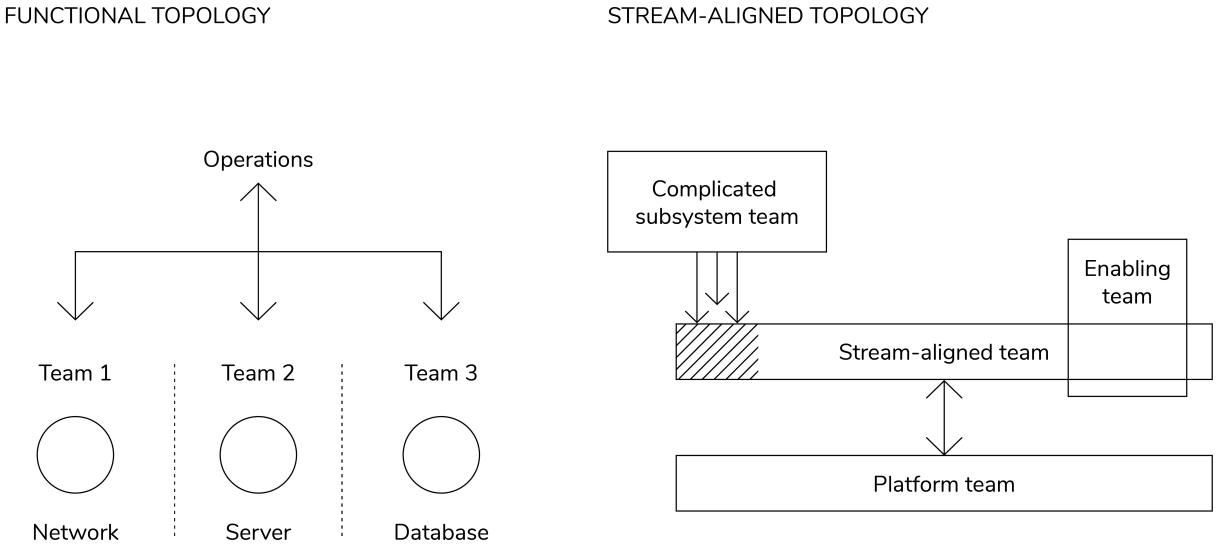

Fig. 12. Comparison of functional and stream-aligned paradigms. Adapted from: Kim et al., 2016: 83 and Skelton and Pais, 2019: Figure 7.5 in Ch.7.

The two kinds are sometimes referred to in DevOps as the functional and stream-aligned team types, and for the present study, I refer to both as topologies, however of two different foci (Fig. 12). On the

[185] Conway, 1968: 31.

left of Fig. 12 is the diagram of the functional type, where the teams work separately based on their functions and report to the operations team, which serves as the mediator of all communications between them. In the diagram example, network, server and database teams are divided based on the components they work on. Due to the explicit separation of operations into its own unit, this configuration enforces the separation between operations and development. The alternative topological configuration, portrayed on the right side of Fig. 12, does not have this separation and is more resilient in the face of change. This is referred to here as a stream-aligned team topology, and mixes the specialisms from different functional units together, having at the core the problem the team addresses, rather than the function team members carry out. The diagram shows the relation between Skelton's main stream-aligned team types: platform, stream-aligned, complicated subsystem and enabling teams. The two kinds of team topologies are discussed in more detail below.

Functional topology

Functional team topology is formal and rigid, which makes it fast, yet fragile in situations of rapid change. It is optimised for either expertise, division of labour or labour costs. In DevOps sources, functional topology is seen as a team arrangement which had historically been dominant prior to the confluence of Operations and Development to form a new boundary discipline of DevOps. The functional topology is enacted through the hierarchy, and groups staff based on their specialisations – for example, server team, network team, database team. The key advantage is that functionally oriented teams can achieve great productive velocity in periods of stability, due to their good traction – a combination of organisational momentum with full transparency to the practices of audit. The team members are capable of processing the tasks quickly when they are used to working with one another, well acquainted with their tools and the tasks they work on are similar enough to be able to process them in a similar manner without drastic changes to the workflow. This orientation can be ideal for technical support in large systems, and arguably, when the velocity and momentum are good enough, changes can be introduced gradually via collaboration with enabling teams without noticeable detriments to the overall performance. The slow responsiveness to change in such topology can largely be attributed to the fact that different specialisms never work together and are located on different fracture planes, which means a lag in their communications.

Some functional topologies deal with this problem by introducing the elements of stream alignment and are usually referred to as matrix topologies. This arrangement is meant to be flexible in that it supports a dual function in carrying out both functional and feature work. However, they achieve this at the expense of somewhat muddled reporting protocols, since individual contributors often have to report to both business and functional managers. In organisation theory, such an approach is usually

frowned upon because when a subordinate has more than one person to report to, this leads to conflicts in crisis situations, since they would have no idea who is taking precedence as the primary line manager.[186] Despite their drawbacks, matrix topologies can be used either in cases when avoiding them can create unnecessary complications, or when using purely functional design is too difficult. In the first instance, the reporting can be doubled but happens in the orthogonally positioned organisational planes with no risk of conflict. For example, when a standard market or functional orientation is supplemented by additional reporting on specific supplies or expenditures. In this case, the work progress is reported to the department line manager and the balance sheets are submitted to accounts. In the second instance, an example could be a team that is assembled on an ad hoc basis to provide a specific service which requires expertise that other teams are lacking. This may lead to conflicts, but it is too difficult to provide a highly specialised service via any of the other two orientations. Furthermore, as Herbert Simon notes in his discussion of functional organisation, the teams have to be split in such a way that any budget decisions are made at the point where it is possible to evaluate different value propositions and select the most cost-effective one, and there is no bias arising from the substitution of organisational objectives for personal aims.[187]

Stream-aligned topology

The stream-aligned team topology is a more recent approach that was theorised by the practitioner Matthew Skelton based on the context of DevOps and Continuous Delivery method. Skelton defines a *stream* as 'the continuous flow of work aligned to a business domain or organisational capability.'[188] This makes him propose stream alignment as a set of principles for designing a technology-based organisation in accordance with Conway's proposal, focusing on the construction of communication within the organisation or the team to match the design of the software system that the company develops. Adherence to streams avoids some parts of the friction which Frederick Brooks warns about in something that is known as Brooks's Law, which claims that 'adding manpower to a late software project makes it later.'[189] In contrast to the functional view, the stream-aligned topology places the priority on associating the company's business proposition to the delivery pipeline, which is its technological core. The stream alignment maps teams in relation to the pipeline in a way that would allow them to continue delivering customer value in situations of extreme uncertainty, avoiding any direct confrontations with software complexity.

---

[186] Simon, 1997: 31 and 191.

[187] Ibid.: 295.

[188] Skelton and Pais, 2019: Ch.5.

[189] Brooks, 1995: 25.

As Gene Kim notes, prioritising streams over products or features creates better conditions for the conversion of business hypothesis into customer value – in this context, via a technology-enabled service.[190] Rather than splitting based on the systems' functions, stream alignment has teams focus on specific aspects of system features at stake: the stream itself, as well as the platforms, subsystems and support services for any of them, and interact with one another as individual service providers. The stream-aligned topology can also be implemented in contexts that do not match the organisation as an entity – for example, in an agency which handles differently organised work, only its selected departments responsible for specific services can be arranged in a stream-aligned way. Streams can also overflow organisations – for example, in open source and other large initiatives, the stream span communities of practice across organisational boundaries. In essence, the stream alignment aims at creating the organisation structure that overlays the system components in such a way that enables DevOps to focus most of their efforts on mitigating the complexity of the relations between components. This requires the teams to be flexible enough to change their positions in the workflow, which in stream alignment is achieved via the service-based treatment of their work. While the team boundaries are rigidly maintained, the internal composition of a team itself is taken more flexibly to allow for rapid adjustments to maintain the quantity and specialisation of team members in one stream relevant to the amount of complexity, and to reduce the efforts associated with hand-offs and integration.

Since the present discussion is interested in the relations of teams to deployment, the other three team types formulated by the team topology framework – platform, subsystem and enabling teams – can for the moment be grouped together as not aligned to the stream, or simply *non-aligned*. This will make the discussion easier since the latter three team types are not a part of the technological value stream. Instead, they provide ad hoc services that concern specific technical aspects of the domain. Nevertheless, a brief reflection on the non-aligned teams' duties will help to illustrate how a confrontation with software complexity can be avoided through the application of organisation design methods.

The platform team delivers the platform, which in this context should be defined as a product or service used by stream-aligned teams to deliver revenue-generating or customer-facing features, yet which is not necessarily a part of the company's value stream itself. A distinguishing feature of a platform is that it is self-contained and provides an API or other interface, documentation and community or customer support. Examples are the Linux and Windows operating systems, the Java Virtual machine, public cloud services such as Google Cloud, Microsoft Azure and Amazon Web Services, or a container platform, like Kubernetes. In terms of its use by the business, a platform is, on the one hand, a layer

---

[190] Kim et al., 2016: 8.

of technical abstraction which diverts the software complexity of the system parts which are not associated with the immediate concerns of the stream-aligned teams, such as networking or infrastructure, into the purview of the dedicated platform team.[191] On the other hand, the platform can also be used as a managerial abstraction to encapsulate more complex team structures, depending on how complex the platform is, and how much of it is delivered by third parties.[192]

Subsystem teams are brought in intermittently to help stream-aligned teams with specialised technical skills that the latter do not have on board. For example, in one of my fieldwork cases, the business required us to implement a search function on their website. This entailed an integration of a plug-in provided by our cloud service, however, our core team, which was a stream-aligned team in that it was busy with customer value features, did not have any capacity for such an integration job. We resolved the situation by assembling an additional subsystem team of two additional members, a DevOps and a back-end engineer, who would focus on the integration task for a limited period of two weeks. The subsystem team members would also be present in the stream-aligned team meetings, meaning that the hand-off efforts would be reduced by the time they are finished.

Lastly, enabling teams are the research units, and even though they are similar to subsystem teams in that they are highly specialised, their role is guidance and not execution. In the example of my field study, one of the business requirements was to implement a scalable cloud architecture, which would allow the production team to add more computing resources to the platform in moments of high traffic, such as in cases of online events or special project launches. This required a new set of skills that the stream-aligned team did not have, and thus we have requested a DevOps specialist to be regularly present in the weekly team meetings for a period of six weeks to report on the progress of scalability implementation and answer any questions, which allowed for a soft transition to the new method of operation.

Additional utilisation of epistemic infrastructure as code (EIAC) in this context lies in its logical splitting into several different pipelines where the circumstances make it necessary. The scenarios generally depend on the team topology, considering the ownership. For stream-aligned teams, different deployment pipelines can be used when the parts of the system belong to different streams. Subsystem teams stream their work as a service that other teams use without having to consider the underlying complexity details of the service. Platform teams can have their own pipelines for specific platforms – this is the

---

[191] Skelton and Pais, 2019: Ch.5.

[192] Ibid.

best practice for deployment for any components that are stable or use distinct technologies or build processes. For enabling teams, separate deployments are not necessary since the ownership rests with the stream-aligned team. The epistemological practice of splitting is particularly important for large systems which deploy great amounts of inventory, by making it possible to audit the relations between teams in a scale-free way, as Chapter 5 explains further.

In its topological sense, a stream establishes a continuity between a specific aspect of a business domain and an organisational capability that the stream defines: a user journey or a user persona, a service or a set of features. While there can be a case when a stream addresses one product or one feature, the conceptual disentangling of the stream from whichever content it deals with creates an abstraction of a continuous initiative. This abstraction makes the production processes available for symbolic manipulation, making it possible for DevOps to strategically engage with services, feedback, failure and learning. Dealing with such matters may otherwise be difficult to do, since the abstractions of functional topology, such as a product or a feature become too fuzzy in the context of stream-aligned production, and may be difficult to rely on.[193] Furthermore, based on the stream continuities, the team relationship shifts from reporting based on products or features to *x-as-a-service,* a term which refers to the service nature of the team unspecific to what the team does. This maintains the rigidity of the team boundaries, but in contrast to functional teams, enjoys a higher bandwidth for circulation of knowledge by supplying each team with an 'interface', or a standardised way of exchanging the outputs of their work. This further contributes to the abstraction of the production process and creates the conditions in which the relations between teams can be addressed separately from the teams themselves, for technical troubleshooting, customer experience, audit, or other mechanisms of governance. In terms of team interactions, besides the x-as-a-service kind which uses one-to-many relationships, with matters of service strictly owned by service providers, the stream-aligned paradigm also offers collaboration and facilitation interactions. These two are defined through the different ownership character. Collaboration is preferable in one-to-one relations and shared ownership, and facilitation is possible between a stream-aligned team and a small number of facilitators, with ownership retained by the stream-aligned team.

The team topology approach could be considered in epistemological terms because it provides the tools for creating, organising and warranting knowledge. Such a position is common to much of the operations research, which tends to focus thinking on how things are done, rather than on the content. The outcome is that content becomes one matter of analysis among many others, including infrastruc-

---

[193] Skelton and Pais, 2019: Ch.5.

ture and delivery themselves. Through decoupling of teams, and providing different types of interactions, the topological approach opens the alleys for more detailed critiques of the continuities that may exist within the organisations and the validity of epistemological claims that may arise within the streams and between aligned and non-aligned teams. Moreover, through its nuanced approach to the organisation design, it opens access to the design of the software system the organisation works on, and for the comparisons between the organisation and the system design. There are at least two major benefits of the topological approach for the present study. One is the ability to implement the delivery pipeline to handle the contingent complexity of relations between the components to be deployed. This also means that the stream-alignment paradigm generally supports and even assumes the adherence to Farley's Continuous Delivery method. The other benefit is that the topological approach is well suited for understanding the parsimony principle of agent's behaviour, which Chapter 5 will look at in more detail, and which implies accessing the individual agents in terms of their neighbourhoods, situated knowledge and involvement with their proximities.

Circulation of knowledge within the production system

The design lifecycle comes into being through a particular way the knowledge circulates within the production system, which this section aims at explaining in more detail. Getting back to the design lifecycle diagram (Fig. 13), it is now a good moment to provide commentary on the process of knowledge inventory circulation between the epistemic infrastructure and the problem space that sets the whole of the design process in motion. On the one hand, the epistemic infrastructure creates the possibility for the action, through creating a blueprint of the problem space via the components. The problem space of production, on the other hand, is where the negotiation of the system's criteria takes place. The three groups of stakeholders, through negotiations, excite and disturb the equilibrium of the system and appear as the axis around which the inventory circulates. To use the terms of compositional methodology, the epistemic infrastructure serves the givens, by providing the next step of problem definition with each new deployment iteration. The goals come through the problem space criteria as they receive new requirements from the business, or the changes in production dynamics because of the change in technology, or new incoming user demands reflected in the customer values. The stakeholder groups, such as production teams, business headquarters and users occupy the middle place as operators, being responsible for defining actions and methods.

To further explain the process of circulation of knowledge within the production lifecycle, this section starts by explaining why the infrastructure as code is always identical to the structure of the organisation, and vice versa. The coincidence is achieved through constructing a more thorough understanding of the differences between the notions of network and infrastructure. Then the section turns to the

ideas of process and product, which throughout the Continuous Delivery acquire a more fuzzy character and tend to blend or collapse into one another. Lastly, the section examines why software systems tend to disintegrate with time. The section results in an important observation that, due to the coincidence between the code and the business value stream, the tendency of such code to disintegrate presents great risks to the organisations, adjacent communities of practice and wider society. The DevOps-oriented study of production should therefore be considered a top priority for software studies and other critical scholarship that deals with operations.

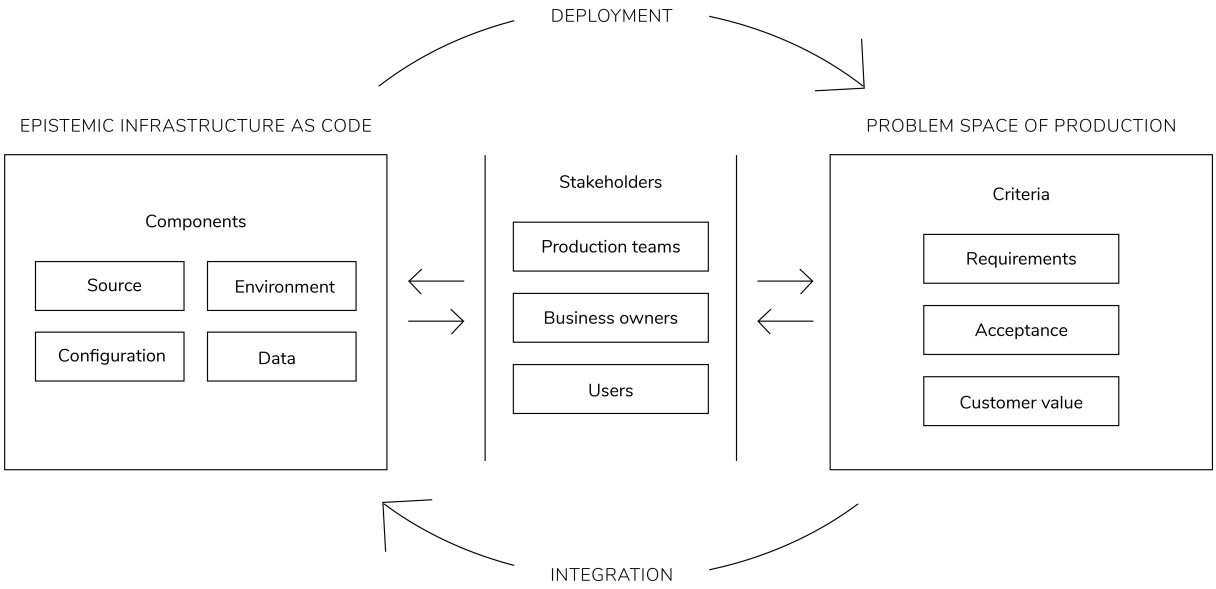

Fig. 13. Production design lifecycle.

The structural coincidence of the organisation and its code

There is a close relation between the terms *network*, *infrastructure* and *system*, and while the term *network* usually stands close to the matters of knowledge management and complexity, going too far into its exploration is beyond the limits of this study. It is necessary, however, to understand the relationship that the phenomenon of the network has with the systematic approaches and the infrastructural attributes of knowledge. In the definition provided by one of the key thinkers of networks, the sociologist Manuel Castells, a network is a set of interconnected nodes, where the node is something that is defined in terms of the network itself. The prominent feature of the network for current analysis is its topological quality, which is relevant to the present research in its three manifestations. In the first instance, the network topology relates the distance between nodes to the intensity and frequency of their interactions, with the nodes belonging to the same network having a more immediate relation to each

other.[194] Second, the power relations within a network also display topological characteristics. The decentralised principle of resource distribution in the network is based on node clustering that generates situated knowledge. Therefore, the power relations are focused on the access rights applied at the nodes because the node switchers and the inter-network relay operators have the most influence on the process that takes place.[195]

Third, network topology sees the non-uniform dissemination of information across the network. Beyond the team configurations we saw above, the other important impact on the properties of the network comes from the associated communities. As discussed by the organisation scholars John Seely Brown and Paul Duguid, communities act to create areas in the network which have their own distinct identity and coordinated practice.[196] Lastly, the networks are not limited by organisations, such as, for example, communities of practice around large technological initiatives of Linux, Salesforce or the DevOps movement in general. This suggests that notions such as organisational culture are not as uniform as they are usually perceived – 'the way we do things around here' might differ in various parts of one organisation, and perhaps relate these parts to the wider cross-organisational communities engaged with the same practices, creating a matrix when overlayed with the organisation's internal structure.[197]

With such a view of the network, the *infrastructure* is a network where the nodes are the institutions, people, buildings and information resources that generate, share and maintain specific knowledge.[198] Infrastructure in this sense differs from the system in that its aim is to create conditions for the activity, rather than the activity itself. Given such a role of the infrastructure in the system, the system, in turn, can be defined as a unity of goal-oriented social and technical principles and processes informed by their infrastructure. There are two features of the infrastructure pertinent to the present discussion. One is that the infrastructure is system-agnostic, that is, can be either applied to specific systems, or different systems simultaneously or in non-system environments. In any case, the infrastructure's role is to inform the strategic formation through a sort of standardised kind of access, a protocol, an API, or another interface. The other feature of the infrastructure is that it does not presuppose its knowledge, as well as the interpretations and understandings that it is derived from, as fixed. Instead, they are

---

194 Castells, 2010: 501.

195 Ibid.: 502.

196 Brown and Duguid, 2000: 144.

197 Ibid.: 161.

198 Lury, 2021: 207 citing Edwards and Bowker.

streamed based on the continuous feedback loop that delivers the new data derived from the application of methods in the problem space of production.

Moving further to a more specific term of epistemic infrastructure, it should consider the meaning of the term *epistemic*, which refers not to the content of knowledge but to how it works, enabling a discussion around the regimes and modes of understanding, interpretation, belief, explanation or justification. Approaching the content in terms of its epistemology allows for the analysis of the ways in which the empirical findings are interpreted, alternative constructions of the objects of knowledge and knowing subjects. Adapting the term for the analysis of production, *epistemic infrastructure* can be seen as an infrastructure that provides a comprehensive narrative about technical, strategic, business value and other aspects required for effective production and maintenance. Moving further into DevOps, the epistemic infrastructure as code (EIAC), which this chapter aims to develop, adds to the definition that rather than a set of operational entities, the infrastructure is instead an executable instruction that grants access to such a set, organised in a way that enables it to create from scratch a software system as a whole.

Based on the above assumptions, it becomes clear why the present study needs to differentiate between a system, a network and an infrastructure. The common feature of all three phenomena is that they denote a limited set of artefacts and concepts which are connected in one way or another. While the construction of the three is similar, the difference lies in their teleological causes, or in other words, the rationale as to why these entities are gathered together. The system describes its components in terms of their functional contribution to a common goal, the network in terms of their topological clustering, and the infrastructure in terms of possibilities of knowing.

It is possible, the present argument goes, to think about EIAC because there is a close link between the infrastructure of knowledge about a software system and the system's configuration which is written, executed and maintained as part of the production process. The link is maintained through the limitation of organisation design, as Conway's Law postulates, by the organisation's ability to communicate what is it that it can do. In other words, whichever pre-existing structure the organisation already has, many of the design alternatives would not be accessible to it, due to the absence of the communication links that could inform of the existence of such possibility. Turning to the epistemic infrastructures of IT companies occupied by the production of a specific software product, the organisation's communication structures would likely tend to become wholly synchronised with the structure of that software product. This means that, on the one hand, all parts that the software needs for its successful operation are written down in the infrastructure code. On the other hand, the organisation's communication is

structured in a way that supports the production of all these parts. For the present study, the two infrastructures can be equated because any part that can be referenced in one infrastructure can also be referenced in another.

Product becoming process

As we have seen, the Continuous Delivery pattern endorses a software system which is not a durable copy or a model but is instead something streamed, that is, delivered from a clean slate, via the deployment pipeline, every time it is called for, along with its configuration, environment and documentation. Such a technique ensures that the software system is always present in its most up-to-date form. While such an ideal scenario is prescribed as the best practice, the emphasis here is on the possibility for it to work this way, without the assumption that this is how it would necessarily be done at all times because the circumstances can prescribe different deployment patterns for different software systems. The continuity of the process, however, causes some confusion in terminology, specifically between the notion of the *product* and the *process*, which bears an impact on identifying which authority decides on the meaning. This is particularly hard to define when the epistemological conditions are produced and made available as part of the software system, removed from the purview of knowing subjects and into the components and access permission schemes.

In this sense, the circulation of knowledge can be seen as happening not between the models and an equivalent of a *physical copy* of an artefact, but more directly between the epistemic infrastructure and the problem space. The focus is shifted because there is no longer a notion of a product as a unique player in epistemic circulation.[199] Instead, there are components – code binary, configuration, environments and data, delivered as a service. In this scenario, the content is separated into the database and the principles of understanding and interpretation are contained in the EIAC, maintained through version control. This effectively disables any epistemic circulation among the environments which come and go continuously, meaning that any knowledge that may have existed between them and which was not reflected in the infrastructure script, documentation or database, gets overwritten with the new release. This, however, does not preclude the sense of situatedness, since each environment may be used through its own distribution schema. To draw from some of Farley's examples, in A/B testing some audiences are presented with a different version of the same product to obtain new usage telemetry; in user acceptance (UAT) or other testing environments the users can be limited to groups with specific access permissions for testing and approvals; in blue-green deployment method, the system changes require switching from one database to the other without losing the changes made to the

[199] Lury, 2021: 147.

database in the process.[200] This way, situated knowledge is still constituted in circulation between the application of methods and reproducible epistemological values, albeit remapped to the layer of configuration scripts.

One of the distribution benefits of Continuous Delivery as discussed in the DevOps technical sources is that the automated streaming of infrastructure delivers a personalised topology of user experience based on the choices made through interactions, for example, displaying otherwise password-protected areas or offering buying suggestions on an e-commerce website. At the same time, the experience is scale-free, in the sense that it is delivered in a personalised way to each user without adding extra work, which is partly realised via relational database lookup, and partly by a suite of automated tests that alleviate the audit practices. The experience is regulated and organised for data collection through testing to be deployed again and again based on the uninterrupted stream of feedback coming from various channels. For the current discussion, this is the *problem space of distribution*, which is hybrid in that it is simultaneously common and not common, and is characterised by its continuities rather than by its discreteness. Such a view, informed by operations research, stands in apparent opposition to the notion taken for granted in a more reflexive style of software research, something which is often used in media theory.

Such a media theory approach focuses on the critique of the events happening in runtime, where the user experiences can be seen as highly differentiated and personalised. For example, algorithmic recommendations on film or e-commerce platforms can be different depending on which user account or geolocation credentials are used. This fact can be drawn upon to substantiate the claim that recommendations and data harvesting in general can be detrimental to the user experience of cyberspace as the commons.[201] The present research focuses on complexity in the deployment stage and thus is not concerned with the events that happen after the binary has been delivered. Furthermore, the complexity of the problem space of distribution, in terms of the present study, can be abstracted away through the notion of the *environment*. As we saw, the environment is a configured set of resources, which encapsulates a specific user and works with the data related to user interactions. In this sense, the environment can be pointed out as a useful analytical tool for further research on software distribution, which would provide a diffractive, rather than fixed, topological view, which is, however, out of the scope of the present study.

---

[200] Humble and Farley, 2010: 261.

[201] Lury, 2021: 90 citing Cardon.

The tendency of systems to disintegrate

The implication of the equivalence of the infrastructure of the firm and the one of the production system is that any organisation design initiative has to account for the communication needs of both entities. The present research is concerned primarily with the epistemological aspects of such communications, that is, in which way the knowledge of organisation is replicated in the infrastructure code. The process of establishing an organisation's communication channels, and concomitantly, the production system's epistemic infrastructure and problem space can be roughly split into two stages. The initial stage aims at understanding the boundaries placed on the system by the stakeholders and the world's realities. Such boundaries can be negotiated on the level of the organisation's value stream and come in the form of a strategy that translates the value produced into the technological solution. The task of the second stage, once the strategy is done in its either preliminary or more detailed shape, is to draw up the tactical team topology. The final topological configuration depends on the design of the software system it is required to produce, its streams and the amount of platform and complex subsystem support the streams are going to require. The resulting pairing of strategy and tactics begins to be carried out in practice once both stages are complete, which faces the organisation with the task of coordination. The coordination is done within the design lifecycle and consists of such activities as maintenance of the boundaries and stream configurations, delegation of tasks and their coordination, and integration of components into a whole. Testing activities in organisation design management are the important consideration as they provide the means to mitigate the risks created by the scarce resource of human attention.[202]

Testing is included in all production stages so that the errors are addressed not only via the formal testing stage, through actively detecting errors, but throughout the production processes, by using such procedures as Test-Driven Development (TDD). Commonly associated with Agile methodology, TDD essentially implies that tests are written alongside the source code. Such a tactic, as Farley explains, makes tests useful not only as executable specifications of the expected behaviour of the code but also as regression tests and documentation later in the process.[203] Dispersing the tests throughout the production process, alongside the continuous dispatch of small but frequent updates avoids the bottleneck of scarce attention since each error is still fresh in the memory and has not been around long enough to create any dependencies that may add more workload to troubleshooting later on. The stream-aligned paradigm sees integration as a risk and aims explicitly at reducing hand-offs between teams by making sure that each team is equipped with team members with different functional orientations to

---

[202] Simon, 1997: 241.

[203] Humble and Farley, 2010: 71.

be as self-contained as appropriate for the continuous stream delivery. The efficacy of such a tactic can be assessed through Conway's three ways in which systems disintegrate.[204] One deals with the fragmentation of communications and is reminiscent of Brooks's Law discussed earlier in the present research. While there is a general tendency for an increase in the staff assigned to the design effort, the increase in potential communication paths increases exponentially with each new member. This may mean that even moderately small organisations have to apply a considerable effort to restrict some communication to make time for teams to actually perform their work duties.

The other disintegration dynamic is that the staff increase happens because system designers choose to delegate tasks when the apparent complexity of the system approaches the limits of their comprehension. Such behaviour, however, goes against best practice, which prescribes that designers either direct all their efforts at reducing complexity or risk giving up control over the system altogether. Real-life situations, however, add time and budget pressures, which confront the system designers with the choice of losing their reputations over mismanagement or deferring the complexity to a later moment in time. This means that complexity is being deferred by expanding delegation, where the tasks are being reassigned without having adequate resources to address the underlying systemic flaws that generate more tasks. Furthermore, the budgets are getting overblown due to the management logic of reliability: if less budget is spent and the effort fails, the administration will be seen as incompetent. Conversely, in the case of a larger budget expenditure, the failure will come as evidence that a problem is indeed a difficult one.[205] In other words, as long as the managers' prestige and power are tied to the size of their budget, they will be motivated to expand the organisation. As a result, this situation makes Conway call for a change in system design management thinking that is not based on the assumption that adding manpower simply adds to productivity.

Lastly, another disintegration antipattern derives from the primary observation that the communication structure of the software system coincides with the one of the organisation, which also means that if disintegration is allowed in the organisation, the system will also lose its cohesion. Since the system's design is not directly accessible, it should be improved by the application of reverse Conway's Law – designing an organisation's communications alongside the natural seams of its software system. Such a requirement comes hand in hand with the call for increased attention to change management because the requirements for software systems may create unpredictable consequences to team compositions and organisation structures. As we have seen, this pressure is rather effectively addressed by the stream-

---

[204] Conway, 1968: 31.

[205] Ibid.: 31.

aligned team topology comprised of longer-term members with broadly applicable sets of skills, who are supported by the ad hoc teams to target more specialised tasks. In systems organised as such, some institutional memory can be retained for smooth incorporation of new cultural features associated with learning and other types of change. At the same time, stream-alignment makes systems better prepared for radical innovation than the more ossified large technological systems Chapter 1 saw theorised by the historian Thomas P. Hughes.

Are any of the three disintegration trends addressed by the Continuous Delivery and stream-aligned workflow? From what the study has learned about these methods so far, it is clear that they have proven to be very effective in streamlining the path from business value priorities to customer value, which seems to resolve the large portions of what is seen as the second and third facets of disintegration. The fragmentation of communications is reduced by organising the teams along the fracture plane patterns so that everyone who has to communicate often has the highest bandwidth – and this configuration is constantly revised so that those team members who communicate the most always stay close as team compositions are adjusted. The coincidence disintegration problem is mitigated indirectly by reversing Conway's Law to design an organisation to coincide with its software system. Despite all of this, however, the first problem of overpopulated design efforts remains largely unaddressed.

The evidence for that is the traces of waste and hardship which are frequently referred to in DevOps professional sources and can be described as the symptoms of complexity which appear in places where epistemic infrastructure and problem space are not aligned. These can relate to either a problem within one or the other, or the relationship between the two. The signs of problems within epistemic infrastructure can be things like *technical debt* – accumulation of work related to the temporary fixes made earlier – partially done work, missing or unclear information or commissioning of extra features which do not necessarily relate to the value stream. The problems of misalignment between the two production categories lead to ineffective behaviours such as extra or manual work, extra processes, waiting or heroics, meaning that teams spend time doing things a hard way.

The argument that the present thesis tends to agree with is that such symptoms are related to the fact that the overpopulation of design efforts cannot be solved if the organisation is approached in its entirety in a centralised way. Yet, the centralised kind of administration should not be entirely disqualified, since, as we saw in Thomas P. Hughes earlier, it is indispensable for the identification of the large-scale patterns and analysis of the important systemic features such as style, momentum and reverse salient. Furthermore, in the case of Frederick Brooks, theorising conceptual integrity led to important practical decisions in terms of high-level planning and repetitive day-to-day operations, leading to im-

provements in performance in a large software firm. This demonstrates that systems thinking is appropriate when designing repetitive interactions for the results that are known in advance. This is, however, rarely the case with software systems, which, as also shown earlier in this study, meant that a systematic approach had led software production from one crisis to another. In the chapters that follow, my study will therefore turn to organisational formations of a different type, referred to in various sources as either distributed or complex adaptive systems. In my research, taking inspiration from Arash Azadegan and Kevin Dooley, the terms are seen as referring to the same phenomenon but for different reasons. Referring to the system as *distributed* expresses its relation to the outsides, for audit and other procedures of governance. Alternatively, describing a system as a *complex adaptive* formation speaks to its internal organisation in terms of its constituent parts, for example, the self-organisation of its agents.[206] In this interpretation, the system is *complex* due to the multiplicity of local communications, and it is also *adaptive* because the shared meanings are being negotiated in conversations that presuppose mutual adjustment of agents throughout the negotiations.

The section should be concluded by mentioning that the structural coincidence of code and organisation combined with the tendency of software systems to disintegrate result in a dangerous trend where the failures in epistemology inevitably lead to critical failures in the business. This trend is of great urgency and has to be prioritised across the whole domain of software scholarship, equally involving industrial research and cultural studies. The reason is that *the impact of operations on software systems, firms, and, as a result, society as a whole, can be of great scale*. The problematics of epistemic circulation are not a matter of abstract theorising that can safely rest in tech blogs or in 'geeky' communities of practice. It is, on the contrary, a matter of critical importance that directly impacts the organisation's value stream because this stream is immediately linked to the organisation's deployment pipeline. As mentioned earlier in the present thesis, the discipline of software studies has so far been mostly promoted by those interested in software engineering and thus largely dealt with the problematics connected to the creation of source code, leaving the issues of complexity in DevOps somewhat unattended. However, it cannot be stressed enough that in the production model described so far, errors in source code would usually only lead to failures in specific layers of deployment. On the other hand, errors in operations can lead to failures across the business as a whole because they are tangential to the organisation's technology value stream, and its failures are directly linked to failures in operations and can rapidly escalate to endanger the existence of entire organisations and software systems, not to mention the harm done to end users. As an example, one of the most prominent high-profile incidents in recent history was the failure in the deployment of Knight Capital, one of the high-street investment firms. As the DevOps

---

[206] Azadegan and Dooley in Allen et al., 2011: 419.

practitioner John Allspaw describes, a $440 million trading loss was caused by a 15-minute-long deployment error – a period of time during which the engineering team was not able to disable or roll back the production service. This led consequently to rapid company closure, however nothing beyond a policy that would advise to 'appropriately control the risks associated with market access, so as not to jeopardise their own financial condition … and the stability of the financial system' could be offered.[207]

Chapter conclusion

This chapter has sought to clarify the deployment process of the software system as part of the production lifecycle, and the additional organisational and technical mechanisms that make such deployments possible. The overall framework of Continuous Delivery appears as the synoptic controlling protocol that demands all of the system's components and procedures to exist in a versioned and fully auditable shape. This is made possible by the falling computation costs that enable continuous streaming of the software system. The streaming blurs the distinction between product and service, yet makes the system more transparent for the practices of audit through automation and repeatability. The streamed components are deployed in layers and include the system environments, all of their configurations, the source code and the databases. The streaming happens between the two waypoints. It originates from the epistemic infrastructure as code, which contains all of the system's confirmed evidence, and completes in the problem space of production, through which the system is made available to the stakeholders. Confronted with the delivered content, the stakeholders negotiate its meaning with the aim to confirm the new knowledge. Once verified vis-à-vis the problem space criteria, the knowledge is assimilated back into the epistemic infrastructure, at which point the lifecycle repeats. The chapter has found that the depreciating computation bears such benefits as the ability to deliver the components in large interrelated clusters of microservices and streaming of updates continuously, rather than through risky larger updates. These advantages come at the expense of an exponential increase in complexity, which makes it necessary to constantly balance the ability to audit with the pressure to valorise complex processes.

In such a situation, a stream-aligned team topology makes it possible to absorb the complexity spikes by rapidly scaling the architecture of the organisation to match the demand for services. The coincidence has to work from both sides: when the system has to expand due to increasing demand, this needs to happen in a scale-free manner without placing increased demand on the maintenance teams. Reversely, when the team has to scale to address a highly complex aspect of the system, it needs to hap-

---

[207] Allspaw, 2013.

pen by way of abstraction, without causing an avalanche of complexity effects across the rest of the system. Furthermore, the governance model remains largely responsible for the complexity increases, due to the reversal of Conway's Law. According to the Law, as the chapter has found, the changes made in the organisational structure tend to seep into the design of the software system, often to unfavourable effect. The complexity risks thus introduced by the multiplicity of interrelations between the organisation, the market and the technological system are the reason why the present thesis emphasises the role of governance in software production studies. While it might not be possible, I argue, to avoid software crises entirely for the reasons discussed so far, the system's complexity can be addressed in large part by thinking about its governance. It is therefore important at this moment to focus on the controlling protocols, and more specifically on the practice of audit, to which the next chapter now turns.

Chapter 4. Audit and the problem space of production

The chapter aims to establish the idea of the problem space in the soft capitalist production model as the place articulated with and through the practice of distributed audit. The demand for distributed audit is present in software capitalism institutions, I argue, due to the high technological complexity of their operations. The operations, in turn, are complex because they inherit complexity from the software systems which they produce or use. For example, the operations of a factory where the sole concern seemingly evolves around the manufacturing of physical objects of finite complexity are still necessarily entangled with the high complexity in its operations whenever they engage with computation-based technology. Therefore, there is an urgency to understand how to control the regulatory mechanisms in complex production situations, or what this chapter refers to, following the accounting theory of Michael Power, as *audit*.[208] Discussing audit contributes a great deal to the understanding of the complexity effects on the system as a whole and to how the production model should be constructed. The chapter's discussion consists of an interlude and three sections. The interlude presents the fieldwork I did, over the four years alongside my PhD study, in my capacity as a digital product lead. It discusses the challenges of simultaneous involvement with the organisation and the software system, the case-based approach to collecting and organising data, and the field implementation of abductive modelling. Lessons learnt in the fieldwork instruct the discussion of audit via three main themes, which are explained in the three sections that follow the interlude. The themes are the notion of collectivity, the phenomenon of the continuity of dysfunction in the production system, and the necessity to think of audit as a distributed practice.

The choice of these three themes in this chapter is underpinned by the key interest of understanding in which way the critique of valorisation of complexity in software capitalism may be methodologically distinguished from other knowledge-based socio-economic critiques. First, the theme of collectivity argues that software complexity can only be tackled by distributing it throughout the system, which, in turn, means a specific balance of coordination protocols communicated via the organisation's management structure with community efforts of a decentralised nature. Second, the continuity of dysfunction

---

[208] Power, 1999: 82.

is based on the observation that the problem space operates through many interruptions and discontinuities, such as the conflicts of interest between the organisation and the market, or the technical conflicts related to innovation or regression – the backward compatibility of components. The dysfunction here serves as the diffractive mechanism of breaking down the problems and partly addressing them, to be able to keep the system in the minimal deployable state.

Lastly, based on the decentralised character of collectivity and the disruptive workflow, I argue that the practice of audit appears necessarily distributed. The distributed nature is exposed here through the concept of a support ticket as a non-linear audit tool, which brings together the criteria of the problem space for the requirements, the acceptance of results and the customer value. The outtake of the chapter is that to be auditable, a complex production system has to be scale-free, or essentially self-similar and based on cases which can be processed through the application of the same scheme, rather than by developing a customised approach for every case. Such a form of system scalability is necessary due to the specificity of the integration of new knowledge on its various levels, and the imperative to grow in scale – the issue caused by the falling cost of computation, which is addressed in the next chapter.

Interlude. Field application

The process of empirical research for this study was not straightforward for a few different reasons. In the first year of my PhD, my involvement with the organisation, JX, had been purely a means of supporting myself financially in the initial period of the study. Yet, the intense involvement with both the organisation and my research over the years has gradually produced a shift in both areas to such an extent that eventually, it became possible to carry out case study research as part of my day-to-day work within the organisation, *de facto* enacting a participant observation strategy. Over the years of my working with the team at JX, my role changed continuously according to my growing familiarity with the institutional ecology, as well as the changes in strategy and staff. To begin with, I was employed as a visual designer, but soon after I joined, the organisation's strategy had been re-pivoted to digital production of their online media outlet, and as part of the changing team, I became increasingly involved in the capacity of creative product lead, managing the work with a rather particular and often abstruse content management system (CMS).

The several infrastructure migrations that I led the system through while working at JX made me think that the mitigation of software complexity should perhaps be seen as a phenomenon which reaches far beyond any particular programming language or CMS, to the organisation that uses it to circulate knowledge and the market that uses it to circulate capital. Since my start in 2018, the JX production

team was growing according to the work we were expected to turn around, and by 2020–21 consisted of two front-end developers, a back-end engineer and two designers. The largest initial problem when we began in 2018 was that the CMS was performing poorly due to the lack of maintenance over the previous five years, and was ridden with bugs. Beyond technical issues, the production process often suffered in the face of poor communication within the organisation, which led to either schedule overlaps, or worse, the necessity to re-make the work that was already done, to different requirements. Working on my PhD in parallel with work at JX, I have decided to focus on learning more about best practices in software production and started a series of annual reports, of which two have been released before the unexpected company closure in early 2022.

In the annual reports, I laid out the main ideas derived from practical work and theory study in the corresponding years. The reports described the data collection and evaluation efforts based on the aims and objectives outlined in the beginning, dividing the work into projects accomplished that year. This helped me to adhere to the model-based abductive logic portrayed in Fig. 7 in Chapter 2 through three phases. In the first phase of research, I created the trial hypothesis based on the organisation's initial request, informed myself about the context and generated the proposal to discuss with the stakeholders. The proposal would outline the rough requirements for development, marketing, editorial and operations, and would contain the product map, wireframes for key functionality, and drafts for the timeline and work breakdown structure. All of these materials would serve as the incoming data for abductive modelling. The next phase would involve negotiations with different groups, stakeholders as well as design and development teams. Some of the major negotiation events that happened in the early stages of my project were semi-structured staff interviews (2019, 2020) and a larger staff survey in 2020. After that point, and as the office switched to remote operation in March 2020, my data collection efforts became more systematic and ongoing. At the end of my employment at JX in March 2022, it consisted of keeping and circulating the minutes for the regular project and strategy meetings with stakeholders, the editorial team and the digital production team, along with the maintenance of existing documentation including e-mails, project-specific reports, wikis, roadmaps and spreadsheets. The final phase of the model work would be the engagement with it during the process of actual production. This phase would begin after the budget and milestones were confirmed with the organisation and any contractors assigned to do the work. While the roadmap would have to reflect the real deadlines, and the project managers would have to submit the statements of work and breakdowns of the work done, the hypothesis would remain a matter of adjustment based on the new evidence that would arise throughout the work process.

A distinctive feature of the empirical work I did at JX as a whole is that it could not be seen as one project because it did not have a clearly defined beginning and end. When I started working there, the

company had its operations going for several years, and when I stopped, the operations were also stopped midway. Therefore, this work could be more appropriately described as a support initiative of an ongoing nature, split into projects of various sizes, or a larger case with smaller nested cases as its integral parts. The use of abductive modelling informed me of some of the differences between projects and cases. As discussed in Chapter 2, projects have a static report-based nature, while cases are more elusive and flexible. The projects are best described in nouns, such as deliverables, metrics or deadlines. The cases, on the contrary, use verbs, and proceed via negotiating, clarifying or investigating. The presence of a project as an entity and casing as the process emphasises the latter's methodological essence. As the interdisciplinary methodology theorist Celia Lury observes, the methods are best approached as doings because then it becomes possible to understand how they are conducted and carried out.[209] While projects are indispensable for delivery and reports, the cases are equally necessary for tracing the differences in how specific parts of work are done, or to be able to understand how the problems are negotiated, and how the completion of projects changes the hypothesis in the organisation's strategy.

Presenting the fieldwork as project-based but done through casing helped address the two challenges. One is that throughout all of the modelling phases, the paucity or lack of user feedback made it difficult to formulate clear enough aims and objectives. Even though we could definitely see the changes in the relations among the team members and the growth of team culture, little has been achieved in terms of measurable business outcomes. For example, if we could produce new software features such as a new version of the navigation interface or a better search tool, we were not able to tell whether they had a positive or negative impact on the user experience (UX). To address this, I directed my efforts at establishing a case for introducing the UX practices, which I was no expert on and was not in a good place to initiate, as some of the case studies in the Appendix discuss.[210] Nevertheless, casing the UX was possible, and had to be done prior to any actual UX work. The other challenge was the lack of documentation, which meant that there was no existing procedure for transforming tacit knowledge into explicit form. For example, much of the research done before the first annual production report in 2020 was gone, apart from some of the unstructured data contained in Trello boards, a project management tool the organisation used for some of the production work. In the absence of documentation procedure, production had enjoyed the faster delivery because of the tacit knowledge contained in the practices of staff members – a mechanism discussed later in this chapter – at the cost of losing traction when those members left the company. Some of the large changes that caused the loss of traction

---

[209] Lury in Lury et al., 2018: 85.

[210] See Appendix, CS4, CS7, CS12.

happened in the areas of the company's operations (the gallery space was closed, and the work became purely online), software architecture (move to public cloud hosting and change in development workflow as described in the 2020 report) and in organisation structure (new digital production department). In this situation, I used project thinking for generating the hands-on documentation about the work being done, simultaneously using casing, in conjunction with abductive reasoning to create the continuous relationship between the existing strategy and the unknown events of the past.

Distributed collectivities

Having briefly looked at the case-based approach that underpins this study, it is now necessary to address some of the issues it reveals. One of the pertinent issues is the phenomenon of the decentralisation of production efforts, which makes the organisation more resilient to the impact of complexity fluctuations. If complexity is not distributed, it acts to dismantle the system. It makes sense, however, to start by pointing out the benefits of centralisation, which may help to understand why it is difficult to apply in high-complexity situations. Turning to the administration theory principles developed by organisational and behavioural theorist Herbert Simon, centralisation should be recognised as a governance principle that limits the decision-making powers to specific parts of the organisation. This necessitates the creation of a hierarchy, in which the subordinate departments are not concerned with comparing and evaluating the competing considerations, and accept the outcomes reached in the higher levels of the company.[211] The benefits of centralisation lie in the increased effectiveness of coordination, expertise and responsibility.[212] However, centralisation is costly and only pays off when the system's complexity has a limit, such as in material manufacture. In software production, however, the high complexity turns centralisation into a liability, revealing such downsides as the duplication of functions because of the limits to the spread of information throughout the organisation, and exponentially rising communication costs.

This appears as a problem because in centrally-organised systems the information is insulated by the hierarchy levels, and the effort needs to be applied to make general information accessible. This activity is not easily scaled and therefore becomes a problem when the general access information is complex and contained in varied resources. Furthermore, there is a somewhat counterintuitive tendency in situations of the scarcity of resources to refer the decision to the higher levels of administration, while it may be obvious that these are less qualified to take decisions because they are further removed from

---

[211] Simon, 1997: 317.

[212] Ibid.

the world of fact.[213] In terms of Azadegan and Dooley, centralised governance brings the most benefit in the situation where the resources and connectivity are scarce and expensive.[214] In software production, however, neither the knowledge inventory resource, nor the connectivity has a limit in the same sense as in material manufacture, which makes the advantages inapplicable, and the new complexity factor renders the centralised control a hindrance. Yet, centralised systems usually neglect local decisions due to the inability to effectively audit them, which results in the lesser ability of the system to cope with uncertainty and rapid change in the problem space.[215] In distributed systems, on the contrary, local negotiations can be prioritised and made operative as part of the overall production strategy. This is possible because of a specific character of collectivity and the ways of negotiating meanings, matters that this section addresses in more detail.

The problem space of production and the community of practice

A *community of practice* (CoP) is a notion theorised in depth by the educational theorist Étienne Wenger, who sees it as a mutual interest group that is engaged in the negotiation of meanings through practice, understood topologically as the complex social landscape of boundaries and peripheries, made operative by the interplay of participation and reification.[216] On the one hand, participation presupposes that only those who take part in practice can enter into a productive relationship with other practitioners. Practice is only present in its concrete enactment, and therefore actual practising cannot be substituted by a mere membership, an interpersonal relationship or geographical proximity.[217] Reification, on the other hand, is present in the sense of 'making into a thing', or treating abstract notions as materially existing. This means that specific notions are introduced into the negotiation space so that they can, in fact, be negotiated.[218] In this sense, the support ticket which is discussed later in this chapter is a reification of the item of work because creating the ticket enables the stakeholders to start the conversation about the different aspects of the problem the ticket is dedicated to. The two processes are mutually enabling, since no ticket can be discussed if there are no relevant participants, for example, developers or users who would be engaged with the problem. Reversely, in the presence of users and developers but without a ticket or other document at hand, there is little chance that any productive discussion of the problem is going to take place.

---

213 Simon, 1997: 320.

214 Azadegan and Dooley in Allen et al., 2011: 428.

215 Ibid.: 426.

216 Wenger, 2000: 50.

217 Ibid.: 74.

218 Ibid.: 58.

Turning to the relationship between CoP and the organisation, the key consideration in clarifying its character is that it is the practice that creates cultural continuity, making it possible for the CoP to go beyond the borders of the institution. As Brown and Duguid note, the ability to extend beyond the boundaries of organisations can be explained by the fact that the 'connections are dense in some places and thin in others,'[219] – that is, organisational culture does not make organisations internally uniform. Furthermore, the organisation may not deal with CoP as one entity, but also on a level of participation of individual members, which suggests a one-to-many rather than one-to-one correspondence, such as in the software-as-a-service (SaaS) model. This also includes cases where the individual developers are employed by the organisation and interact with the software system according to the organisation's strategy and policies, but are at the same time present as the members of the CoP for that software system alongside those developers who are not employed by the same organisation, just as long as they satisfy the participation and reification criteria – that is, they practically engage with the system and address its abstract aspects through the concrete problem-solving events.

Likewise, despite the repertoire of specific routines, words or symbols which a CoP may share with the organisation, it may not always fully coincide with it in all of its activities. This case is made quite explicit in some of the more community-oriented SaaS platforms such as Salesforce, a customer relations platform that pioneered the model, and by widely used open-source products, such as the Linux operating system. The community-building practices in such cases are robust enough to encompass all of the user body of the system, regardless of their organisational affiliation. For example, the Linux CoP interfaces with every organisation that engages with the operating system in different capacities. As Wenger notes, there are CoPs comprised of specialists in one area of expertise who work in different units but stay in close contact, or the interest group around an emergent technology which has enthusiastic followers in competing businesses.[220] For example, in my fieldwork, the digital production team was functioning as a self-contained unit separately from the editorial team. This divide was well-pronounced and yet felt quite organic, having no negative impact on the integrity of the organisation as a whole. The productive relationship within the organisation as the meeting place of different CoPs was precisely, I contend, because of having to adhere to common organisational goals. This might suggest that it is largely the presence of the organisation's strategy and policy that acts as a trigger of the coordination effort required to create the problem space of production, which would always emerge in the intersection of CoP and the organisation.

---

[219] Brown and Duguid, 2000: 144.

[220] Wenger, 2000: 119.

Furthermore, the collectivity of production in software capitalism should not only be evaluated in terms of the compliance of its work-related activities to the organisation's regulatory mechanisms but also in terms of its contribution to closing the gap between the technological and business value streams. It is this latter dynamic which makes the collectivity create, through its interrelations, the *problem space of production* (Fig. 14). This happens, I argue because in this junction the collectivity is simultaneously accountable to the system's technical functioning and the organisation's audit. On the one hand, the space includes business owners, production staff, users, and broadly everyone who uses the software or contributes to the process of its production. This CoP is activated through professional conferences, online social tools, or technical support platforms that link different stakeholder types together – for example, production teams link to users through support requests. On the other hand, the CoP interactions in the problem space of production are limited due to the coordination efforts that come from the organisation to answer specific organisational goals, which arise via the requirements, acceptance criteria and customer value.

Such constraint makes it possible to use this space to focus the negotiations around the specific production goals, converting the intersection into a problem space. The negotiations can be difficult due to the earlier observation that the shared knowledge about the software system tends to align with the software system's own CoP and not with the organisation's CoP. This can be made explicitly, as, for example, is the case of Linux, which results in the cultural fabric of a specific organisation not fully coinciding with the software system's problem space. At the same time, the organisation cannot entirely rid the problem space from CoP because beyond its participation, in Wenger's terms, the community plays an active role in the reification of the criteria, such as requirements or acceptance. This makes it necessary for the negotiations in the problem space of production to continuously navigate the mixed panoply of interests coming from both the community and organisational cultures.

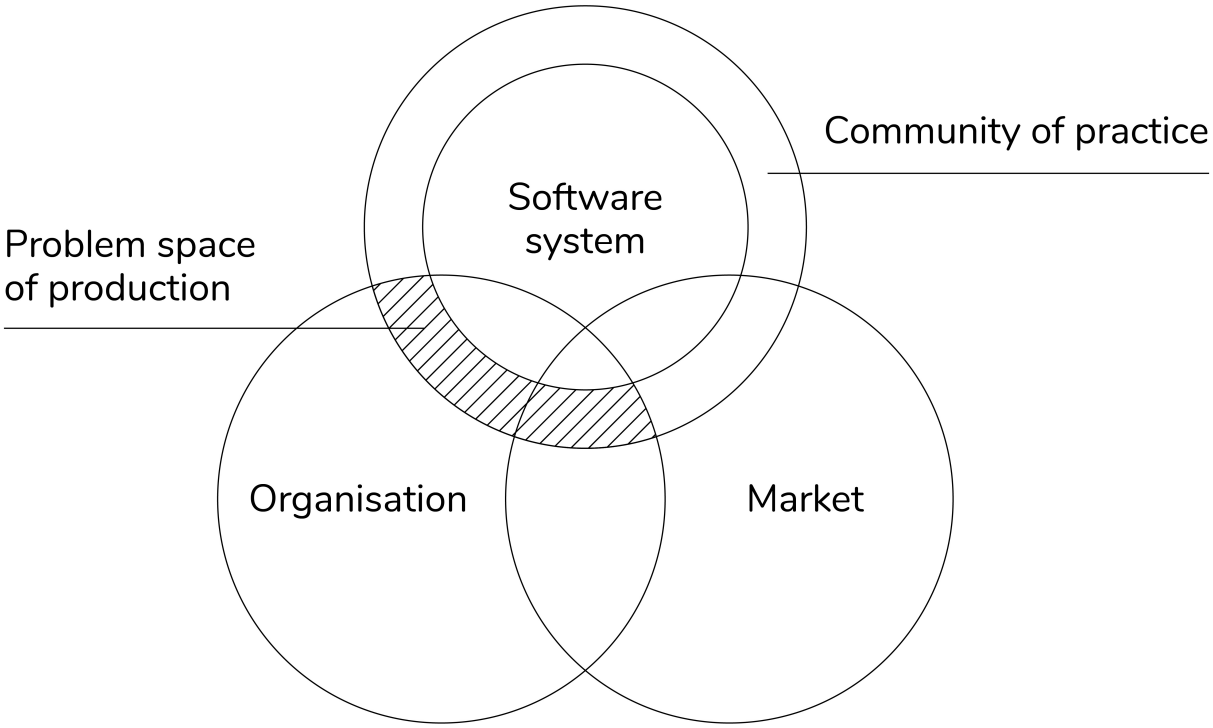

Fig. 14. The problem space of production and the community of practice.

What follows from the presentation of the mutual involvement of the organisation and the CoP is that there is a boundary that marks a part of the problems space which falls into the area of the operations of capital, outlined by the 'Market' circle in Fig. 14, where the activities of the organisation and CoP acquire specific dimensions in terms of their capacities as the forms of machinic and human labour. In knowledge-based production, the positions of such kinds of labour differ from the Marxian capitalist model: the machines are no longer uniquely *dead labour*, the artefacts of investment and the invariant executioners, but the active participants in the organisation and processing of knowledge. Living labour, in turn, delegates some of its cognitive duties to the machine, rather than merely acting as the technical system's appendage concerned with overseeing its functioning. The human and non-human participants of production are grouped according to the current configuration of the problem space, considering counteracting and supplementary tactics. What this means is that the collective no longer benefits the workflow in the Marxian cooperative sense of *a free gift to capital*,[221] or an extra benefit of higher productivity. Instead, there is a capitalist tendency to institutionalise and valorise upon any discontinuities, as this chapter discusses further on. The discontinuities, in the form of the continuity of dysfunction, originate in the problem space of production among the community's goals, which are linked to the process of learning, and the organisation's primary aims are to achieve the clear and transparent structure of outcomes available for audit. In this sense, the conflicts of interest in the or-

[221] Marx, 1990: 451.

ganisation's involvement with the CoP and the software system make the organisation's pressure to audit a more urgent and simultaneously a more difficult task. This calls for an important shift in the way the work is carried out, which can be discussed in terms of differences between the cooperation and coordination mechanisms.

Cooperation and coordination in distributed collectives

Operations treat the organisation's notion of collectivity seriously because such a notion bears a great impact on the wider strategic considerations. Having learnt the lesson of the *tar pit* of early software production systems design, current DevOps thinking appreciates, as we saw in Skelton earlier, that communication is costly and warns that any team collaborations that do not have any explicit business value are to be avoided.[222] The two notions that usually stand side by side, cooperation as the property of collectivity and coordination as a kind of governance, may involve analogous interactions, yet reveal different organisational patterns. Cooperation views the participants of a particular activity in terms of their common goal. Coordination, in the description of organisation theorist Thomas W. Malone, is a protocol that defines the flow of knowledge between the participants that reduces any friction that may arise in cooperation, with an emphasis on managing dependencies.[223] Without coordination, that is, in the absence of strategic awareness of individuals about each other's actions, no shared goal can be reached.[224] Furthermore, coordination is preferred over direct supervision by audit in high-complexity production because the latter requires an increase in coordination and audit efforts whenever the number of tasks to supervise is increased. Conversely, whenever the activities of agents are isolated and the workflow is split into the execution of tasks and their coordination, the reporting on the results of coordination can be standard regardless of what kind of differences there might be between the actual tasks it reports on. Once the unified reporting is achieved, the practice of audit becomes scalable – that is, does not increase in volume when more tasks are added and can be distributed – that is, can be itself standardised, fragmented and carried out locally.

In terms of risk assessment of reporting compliance, Malone identifies the three main types of dependencies that require coordination. Pooled dependencies present the least risk because they share a common resource but are otherwise independent, therefore can be evaluated in bulk based on their parent resources. Sequential dependencies are intermediately risky, since here the downstream activities cannot be executed prior to the upstream ones, and therefore some manual sorting and grouping

---

[222] Skelton and Pais, 2019: Ch.8.

[223] Malone and Crowston, 1994: 90.

[224] Simon, 1997: 81.

might be required in cases where some activities are blocked while others have already been completed. Reciprocal dependencies are the highest risk because here the activities require inputs from one another, and therefore a case-by-case assessment is inevitable. For situations where too many reciprocal dependencies create a bottleneck, organisation theory proposes creating a custom coordination mechanism – for example, mutual adjustment, which gives up hierarchical control in favour of letting the agents self-organise locally in order to get rapid responses. This reduces the risk from the complete absence of results to having to deal with potentially non-compliant results.[225] Further challenges to coordination and audit are related to the assumptions carried over to DevOps from operations in material manufacture and are addressed in more detail in Chapter 5.

Situated knowledge has a double role in collective practices. On the one hand, it may create great increases in output, such as in the division of labour in the widely cited example of the pin maker which the economist Adam Smith opens his foundational Wealth of Nations with. According to Smith's estimations, working alone, the pin maker is only capable of producing twenty pins each day. When the constituent processes are distributed to different workers, it becomes possible to produce over two hundred times more pins.[226] To Brown and Duguid, this happens because it is easier for individual workers to develop new ways of implementing productive improvements to their respective parts of the work, in other words, generating useful local knowledge, when they are no longer distracted by the constant cognitive switches. On the other hand, however, situated knowledge may reinforce the divisions that help create it if different agents or groups can continue to develop their ways of working within the boundary of the components, as long as the components retain general compatibility with the workflow or the system as a whole.[227]

In the context of factory production, the workers are brought together by capital in labour relation as executioners of clearly defined repeatable tasks and maintainers of machines. The function of coordination is to plan and manage the scaling of the model, which can be done in a predictable way since the addition of humans and machines does not change the equation and brings about the corresponding increase in productivity. In other words, the Taylorist scientific management model with its tendency to make all of the production knowledge explicit down to the recording of every minute detail of every movement, tends to downplay the situated knowledge, which is something which is not always recorded. Knowledge-based production tends to re-introduce this tacit type of knowledge back

---

[225] Malone and Crowston, 1994: 133–114.

[226] Smith, 1976: 18–19.

[227] Brown and Duguid, 2000: 153.

into production, which raises the degree of interaction and delegation between human and non-human agents and introduces a risk of unpredictable fluctuation of the production system's complexity, and with it the amounts of cognitive pressure on individual agents. This, in turn, makes scaling knowledge labour less of a straightforward task, and the function of coordination shifts its attention from the individual movements to binding the collectives together with the web of local connections flexible enough to be able to reshape in places where the complexity of the production situation becomes unbearable. Through notions such as cognitive load, which will be discussed in further detail in the next chapter, it becomes possible to estimate the capacity of human minds to process information and to decide on the system's design. While the design of the system is possible, what is the mechanism that supports the system in achieving this traction, different from the momentum of large technological systems described by Thomas P. Hughes? To understand the new sense of traction, a formation of a specific relationship within the production system needs to be distinguished, the *transindividual* which recreates the sense of cooperation lost in the dissolution of the individual.

Cooperation, the transindividual and traction

As the software production system meets the organisation and market domains, the increase in production scope goes hand-in-hand with the increased pressure for research, learning and conversations. These are cultural activities in the sense that they cannot be compressed in time by breaking down into constituent parts, executed in parallel or optimised otherwise. Instead, such labour requires facilitation by staff that contributes to value exchange only in the second order, with the primary objective being the articulation of the problem space criteria and the creation of the problems to be addressed. Active hiring of additional members of staff into production teams to carry out these functions is not therefore sunk costs, even though they do not create the software artefacts as the result of their labour. Discontinuity is no longer a benefit, as it is on the factory floor, where the break between the workers and the managers is crystallised in the rigid workflow, designed to maximise the benefit of cooperation in the form of increased output of standardised items.

The notion of the *transindividual* points to a moment of collectivity that enables the force of cooperation in complex production contexts. It comes as a result of effective coordination within the problem space of production and yields increased traction due to a more tight coupling of activities of team members. This, in turn, increases team velocity. The transindividual is a product of a collective coming-together of the pre-individual qualities, such as emotions and affects, that is not fully differentiated or integrated into the shape of any concrete artefacts to become the matters of market exchanges. Technology openly enters transindividual formations, since it does not merely offer itself as a tool, but conditions the work process, enters the affective relations and is malleable to the requirements of the

users, which also includes production teams. Such openness within the transindividual creates sufficient traction for triggering cooperation in complex production systems.

The transindividuation tendency in the problem space of production is more than the mere absence of the conflict of interests and is characterised by the many overlaps with market relations of exchange. For example, in market competition, a common goal may lead to something similar to cooperation, even though no transindividual relation between the competitors is evoked. When a manufacturer and a farmer have a common goal, a bargain can be struck, but the relation is still interindividual, evolving around a single point of determining the quality of goods exchanged.[228] And vice versa, inside the organisation a market-like relation can take place, as in the stream-aligned pattern of *x-as-a-service*. The deciding factor here is the aims that the parties pursue: within the organisation, the services are provided to achieve common goals that act to further develop transindividual collectivity, while in the market the interindividual relation is more suitable for the event of coming together to exchange the outcomes of independent productions with the goal of achieving monetary gain. In the third case, as my fieldwork demonstrated, developing a long-term working relationship with contractors tends to stack the market and organisation relations along the organisation's communication seams. On the level of communication within the production team, the relations were based primarily on affect and bodily production. To use Barad's terms, production was made possible via the 'agential capacities for imaginative, desiring, and affectively charged forms of bodily engagements'.[229] On the level of creation of value, the outcomes of production served as evidence of the work done, and operated as matters of exchange and payment for the same contractors who otherwise were transindividual – that is, not present on the market as the pre-constituted individuals providing a standalone service.

Due to these factors, the production of transindividuality may appear in the organisation in a contradictory way, where on the one hand, the organisation needs the pre-individual components to produce the cultural cohesion, while on the other hand, it necessarily rejects the tacit knowledge the transindividual relation is based on, as an impediment to control. DevOps clearly gains from the affective dimension of the relations of production which employs interpersonal connections to develop organisational cultures. Yet, the organisation as a whole may express resistance to transindividuation up to the point of hostility. The reason is that the transindividual is a multifaceted more-than-human entanglement of company staff and technology, which develops high velocity due to the activation of tacit knowledge. The latter, as Chapter 5 explains in more detail, is the knowledge which is inchoate and

---

[228] Simon, 1997: 156.

[229] Barad, 2015: 388.

scarcely communicated, largely present within the production practices in a state which is uncodified and unavailable for easy transfer. The presence of knowledge as tacit does not always mean that it is intrinsically incommunicable, but rather that its custodians are interested in keeping it tacit out of self-preservation. As software scholars Matthew Fuller and Andrew Goffey observe, if the knowledge is not articulated, 'it is because it can only be so at the cost of calling into question the social structures that it supports.' [230] Knowledge can also remain tacit because all the parties share the same context and therefore no further explanations are necessary.

The consequence of the tacit knowledge risks to the organisation's relation to the self-organised transindividual formation is therefore two-fold. On the one hand, transindividuality makes it possible to create momentum where otherwise Brooks's Law prevents it from happening, being stifled by the complexity of communications. On the other hand, despite the rise in momentum, the overall system traction tends to fall, due to the transindividual autonomy which resists control and tends to conserve and guard the knowledge it generates. This contradictory relation is addressed, to a variable degree, by the specificity of production workflow, which the next section discusses as based on the continuity of the system's dysfunction. Such workflow tends to fragment the work into the smallest possible fractions and therefore makes it possible to prevent transindividual relations from completely diverging from the overall organisation strategy, while still being able to employ affect to mitigate the complexity that arises from the dysfunction.

The continuity of dysfunction

Since the audit finds the dysfunction at the end of each of its events of self-reflection, it necessitates the problem space as a space where the dysfunction can be described and processed. The problem space, therefore, stands outside of the epistemic infrastructure, which is not intended to account for the system's errors and describes it in the assumed ideal state. The continuities between the particular characteristics of the problem space and the epistemic infrastructure have to be maintained. On the one hand, there are errors negotiated in the problem space by the stakeholders through the requirements expressed in the support tickets. On the other hand, there is existing knowledge contained in the epistemic infrastructure, which by its nature cannot be sufficient to inform the problem solutions, and requires abductive reasoning about problems that would mitigate the scant knowledge resources. Whichever tactic is used, the demands for audit grow progressively stringent, since complexity spikes tend to aggravate the situation of uncertainty.

---

[230] Fuller and Goffey, 2012: 128.

The compromise that the problem space offers is the fragmented workflow, which redistributes the efforts across time,[231] which alleviates the cognitive load by allowing to defer providing solutions to all aspects of the problem at once. This is achieved by abductive manipulation, frequently referred to as model-based, which requires creating an external representation of a problem, which is followed up by breaking the problems into smaller parts. Once the parts are created, the organisational administration can follow a protocol to sequence and coordinate the dependencies between the parts. For the audit, this is also a preferred way of treating complex problems, since it creates a stable and scale-free body of records about the work done, that can be reviewed in a consistent and systematic way. The two remaining sections of this chapter are interested in examining more closely this process of treating complexity. This section looks at the phenomenon of disruption-oriented workflow that accommodates the continuity of dysfunction. The next section turns to the audit as a distributed practice of review and control.

Production of software dysfunction

The phenomenon of the continuity of errors in the problem space of production largely builds on the notion of a *desiring-machine* offered by the philosophers Deleuze and Guattari. To them, the *desiring-machine* is a defining component in the mode of capitalist production, which is present as a plane that accommodates the forces and agents of the specific production context. Unlike technical social machines described in Marx's industrial capitalist model, desiring-machines are involved in production on a wider scale and include both affective relations and the material aspects of the world – that is, production is immanently present in all aspects of reality. The capitalist model is involved with the desiring-machine as the key organising principle of its components and the consequent relations, and also as the defining logic of the plane itself, which therefore becomes its plane of internal operations – or the capitalist plane of immanence, to use the term of Deleuze and Guattari. Coming from the philosophers' theorisation of desiring-production, I take note of the three principles that make such production distinct from the preceding industrial manufacturing model: the surplus value, the movement of interior limits and the antiproduction. Examining the conjunction of these three principles will help my thesis to articulate the valorisation dynamic that ultimately leads to complexity spikes in software production systems.

The first principle is that surplus value is not merely a difference in quantities between matters that are otherwise equivalent in their commodity status, as in industrial manufacturing. Instead, the surplus of desiring-production is created through the differences in heterogenous magnitudes, which come in hu-

---

[231] Hutchins, 2000: 354.

man, monetary and machinic forms. The human surplus value is created in the fracture between the flow of capital and the flow of living labour, the monetary surplus value arises from the divergence of flows of financing and payment, while the machinic surplus comes from the lag between the market flow and the innovation.[232] Using this optic for the discussion of the high-complexity production situations makes it possible to recognise the aspects that make it distinct from the traditional accounts of mass manufacturing. One aspect deals with the diminishing relevance of the class relation of labour between workers and capitalists and the increasing role of valorisation on the complexity effects, which create the fracture that has to be negotiated by the stakeholders in the problem space of production. The other aspect is that the valorisation schema builds predominantly on circulation and distribution, intensifying the monetary flows based on the various moments of the system's disrepair. Lastly, there is the pressure of keeping the system's control in balance with the complexity of its technology, so that the traction is not lost.

The second principle is the movement of interior limits, which is based on the notion that the plane of immanence defines the contours of the terms on which the production problems are negotiated. If an unprecedented problem arises, or a previously unfamiliar feature of an existing problem is encountered, the contours are adjusted correspondingly to include it and rebalance the relations across the whole desiring-machine. This tendency of the sphere of capital circulation to expand is consistent with the Marxian premise that any creation of surplus value at one point requires the creation of surplus value elsewhere for which it could be exchanged. This, therefore, becomes a precondition of capital to either continue to seek direct expansion of production or to create more points of production within the existing circulation chains: 'every limit appears as a barrier to be overcome.'[233] In desiring-production, however, the flow is primary. Informed by the flow, capitalism employs the changes that are introduced into the system, to define both the new categories and the materials for categorisation. In the case of software capitalism, the present thesis generally sees the changes as new requirements, acceptance criteria and customer values which have as one of their primary causes the falling cost of computation. The changes create complexity spikes proportionate to the gaps of traction that they create, where the administration model is becoming inconsistent with the new production capabilities. The absorption of changes may appear in the form of new long-term and high-budgeted projects, reorganisations or migrations to new infrastructures, depending on the magnitude of complexity and the size of the firm. The creation of new branches in valorisation schemas in response to the changes allows the firm's production cycle to balance the production method and the business value stream.

---

[232] Deleuze and Guattari, 1983: 372.

[233] Marx, 1993: 408.

The third is a principle of antiproduction or production that bears no outcome beyond the reinforcement of the capitalist production model and exists solely for the realisation or absorption of the surplus. As a result, the way of negotiating problems operative in the desiring-machine is effective only in its appearance, since it aims for no more than creating the schemas for valorisation which, once established, are used to re-create the world in the image of the abstract value-form they reflect. Like the circuits printed by the high-precision plotting equipment on the surface of the Gordon Moore's Intel microchip, the capitalist value schemas appear as patterns on the plane of immanence – and likewise, the more tight and complex the pattern is, the more saturated with valorisation potential the schema becomes. The main outcome of the desiring-machine operation is therefore a production blueprint that continuously seeks to increase in its complexity, with any associated products or subjectivities being its mere side effects.[234]

Such an antiproduction dynamic may suggest that complexity comes about in the moments when the system breaks down. The breakdowns rarely mean the break in all of the system components, but rather that some of the components either prevent the rest of the system to run in production or disable some of its parts, which do not affect the deployability of the system as a whole. The antiproduction circuit employs the disruptions in the circuit to create additions to the system design that bring the system to the operational state, or a state of better repair, yet create more opportunities for dysfunction, since they make the system more complex. The dysfunction is therefore continuous, in the sense that any system's dysfunction is always linked to another one, which warrants that some part of the system will break again. The dysfunction-oriented workflow is something that makes desiring-machine different from mass manufacturing. To Marx, technical machines can only work and produce surplus value when they are in a good state of repair. As Deleuze and Guattari explain, the machines stop working not when they break down, but when they wear out.[235] The machine will be purchased at the specific cost at the beginning of its operation, then operates for as long as it is necessary to transfer its value to the products, and stops the operation when no more of its value is left to transfer. If the machine stops because of the breakdown, its operation is not disrupted but merely deferred because the associated maintenance efforts are assumed to be in place to ensure that it comes back to its operational state and resumes the value transfer.

A desiring-machine, on the contrary, is expected to break down continuously as it runs. Deleuze and Guattari observe that machines of this kind feed on the contradictions which are the inextricable part

---

[234] Deleuze and Guattari, 1983: 31.

[235] Ibid.

of their functioning since they 'make a habit of feeding on the contradictions they give rise to, on the crises they provoke, on the anxieties they engender … there can be no death by attrition. No one has ever died from contradictions'[236] In other words, having encountered the gap or rupture, a desiring-machine relays the maintenance of it to a separate service or other organisational construction, which creates an additional capital circulation loop and generates more of the heterogenous surplus value of human, monetary and machinic means. The desiring-machine, therefore, is primarily concerned with topological problematics, of connectivity and borders: it defines the connections between the parts it creates and reinforces the lines of communication. By doing this, the desiring-machine, intentionally or not, instructs the design of the organisation itself, since as per a reverse Conway's Law, the organisation cannot break the limits or communication divisions set up by the software system it runs on.

The outcome of such an operation to software complexity is that it's being deferred to and distributed among the many parts of the system, which makes it possible to adsorb and survive its tremendous thrust. While the system's complexity introduces contradictions, such contradictions do not act to completely destroy the system but rather create the stumbling blocks or bottlenecks that deter the functioning of the system's various aspects. The dysfunction means that the constituent parts of the system are not entirely consistent with each other, and in fact, full consistency, as it is prescribed by the system's epistemic infrastructure, is not achievable in the problem space. This happens because, even though the deployment is continuous, the definitions of the problems have to be negotiated all over again every time the agents come together to review the system's present state. The inconsistencies that cause the dysfunction are continuous. Furthermore, fragmenting problems into smaller parts aids the negotiations because the large problems may contain multiple internal dependencies that may cause conflicts if a problem is approached as one task. Each constituent sub-task can be addressed in a focused way via a support ticket, where the requirements, acceptance criteria and customer value are clear enough and can be assessed side by side.

As I found in my empirical study, the discontinuity that necessitates the restarting of negotiations sometimes means that parts of old production knowledge are easier to simply give up. Generating the knowledge by looking at the support ticket archives and technical documentation yields knowledge which may often be obsolete. Appendix case study CS15 gives an example of such a route organically taken by the new developers in the absence of the previous members of the team. Learning by doing, even though the previous knowledge was amply documented in the wiki pages and the support tickets,

---

[236] Deleuze and Guattari, 1983: 151.

did not affect team velocity, and on the contrary, helped to identify the code inconsistencies and reduced the regression testing and refactoring efforts.[237]

Besides being a staple component in the topology of the problem space, the support ticket is also compatible with the system's EIAC. As a part of the infrastructure, the ticket warrants that the problem solutions can be integrated back into the system, subject to regression and other testing procedures. In summary, the deployable system is never a fully working system upon closer examination: it inevitably contains a backlog of bugs to be fixed, it comes with technical debt in various stages of servicing and with refactoring procedures that address that debt. The system is therefore defined by its dysfunctionality, is enacted by it – the system's dysfunction is essential to its ability to function.[238] Furthermore, the very capability of the system to fail is one of the basic mechanisms of software capitalism, which uses the attrition processes to build up the rationale for new cycles of production, and the contradictions are precisely the places of opportunity where the sources of surplus value could be found. Given the direct reliance on change to cause complexity, the continuity of dysfunction should be regarded as one of the key valorisation mechanisms of software capitalism.

Distribution of complexity in disruption-based workflows

Discussion of the theory above brings the practical implementation of the software production system close to the desiring-machine. It is necessary to view these two notions together because it would clarify the meaning of the flows associated with the problem space of production, which lie at the core of both the deployment pipeline and the integration processes. There are two incoming flows: one that deploys the components, making them available to the users, and the other that brings the stakeholder requirements, results of acceptance testing and the evaluation of the generated customer value. The outgoing flow consists of new knowledge, such as the results of problem negotiations and the outcomes of applications of problem-solving methods which had resulted in any correctives to the problem space. The acquired knowledge flows towards the system's epistemic infrastructure, where it is appropriately categorised and stored. The other quality of the production system that puts it close to the desiring-machine is its capability for automatic topological production. This simultaneously computational and spatial principle of production, summarised in Chapter 3 as the *topological machine* now needs to be examined through the mode of its operation. The specificity of its operation lies in its ability to balance the dependencies between the smaller parts of larger errors in such a way that the system is maintained in just enough repair to qualify for deployment to a production environment.

---

[237] See Appendix, CS15.

[238] Deleuze and Guattari, 1983: 151.

Furthermore, the desiring-machine dysfunctional characteristic applies across the whole of the production system, being equally present in the problem space of production in the same way as in the epistemic infrastructure as code. The sociologist Susan Leigh Star, in her research on infrastructure, provides some of the field findings pertinent to the present case. Because the infrastructure is complex and consists of many layers, it has many different local interpretations and is not readily open to changes *from above*. 'Changes take time and negotiation, and adjustment with other aspects of the systems are involved … There simply was no magic wand to be waved over the development effort',[239] Star observes, acknowledging that the infrastructure can never be fixed by major global efforts, but only locally and via small increments. Such a constant partial state of repair, I argue, could be thought of as the *minimum deployable product* (MDP), a reverse side of the *minimum viable product* (MVP). MVP is frequently assumed in DevOps sources as the system's initial stable state that can be used in the production environment by test-driven development. MDP, reversely, marks the product's final stable state, which only exists in production until the underlying errors or technical debt of the ideal version of the system as it is present in the EIAC are dealt with. In other words, to maximise the effectiveness, which it sees as the amount of social value relative to the limit of the available resources, *DevOps has to necessarily assure the system's breakdown*. Seen in the context of the disruption-oriented workflow, such a paradoxical way of achieving effectiveness is in fact nothing but the correct operation. Optimisation work that is frequently discussed in the technical literature on DevOps is supported by the rationale of desiring-production to generate the human, monetary and machinic surplus value, and therefore results in introducing higher complexity.

In terms of cognitive disruption, the two key stress factors of DevOps practice are the uncertainty of outcomes and the inability to audit. The problem of uncertainty implies that in complex production situations, repeating the same process may not yield the same result.[240] This leads DevOps to call for abandoning the checklists and the other practices of repeatable audit procedures that aim to prevent making errors. Instead, errors are embraced as the essential attribute of the complex environment, and the focus is placed on the procedures that allow DevOps to address errors, rather than prevent them. The causal mechanisms that make such environments operative trace their historical lineage from the Toyota Product System, which had pioneered the approach through such principles as revealing the inconsistencies in the workflow, creating focus groups for rapid knowledge acquisition around the problems to be formalised and company-wide broadcasting of solutions to common problems.[241]

---

[239] Star, 1999: 382.

[240] Kim et al., 2016: 28.

[241] Ibid.

The second problem is related to audit and lies in the fact that no amount of effort spent on inspection, approvals and quality assurance can decrease the likelihood of future failures. In fact, according to Gene Kim, the additional checks may increase it.[242] In one case, the repetitive quality assurance procedures were proven to create errors if not automated and instead carried out manually by human staff. In another case, requiring approvals from too many team members from different positions in the organisation had created additional work for most of the approvers, since they were not always familiar with what they were being asked to approve. In yet another Kim's case and also in my fieldwork, creating excessive documentation only made understanding the system harder, since it added to the workload of those using it.[243] Furthermore, my empirical study has confirmed one of Simon's observations that performance reviews are frequently absent where the actual production function is difficult to determine, particularly in not-for-profit organisations.[244] In my cases, the difficulty could have been explained by the lack of sales data, since the organisation was not dealing with any sales, which, however, meant that most of the decisions were largely taken without evidence necessary to validate them, and thus at risk of being inconsistent or arbitrary.

The role of representation in abduction

Production of complex artefacts requires compound abductive manipulation, which is too complex to be done without the support of external representations. The abduction that employs such representation is sometimes referred to as model-based and is frequently practised through various forms of visualisation. This section looks at the method of story mapping as it emerges in software production, and plays a two-fold role in the production process. On the one hand, it is a type of diagram which is appropriate for the collective manipulation of the various aspects of multiple complex problems by mixed groups of different stakeholders pursuing different aims. Using the diagram as a mediating device productively breaks down the problems and formalises the support tickets, which are used throughout consecutive production efforts. On the other hand, the story map is also a communication device which works by creating conversations that otherwise might not have been possible. The idea of a story behind the story map implies that each of the parts of the software system has a user story, which, if told right by the production team, is going to be appropriately enacted by the user. In this way, the story becomes an important building block of the production system that sees the teams and users interacting to achieve specific organisational goals. As Étienne Wenger contends, the story works for the recipients through its ability to offer meanings that the recipients appropriate. The story, in an

---

[242] Kim et al., 2016: 32.

[243] Ibid.: 33 and Appendix: CS10.

[244] Simon, 1997: 265.

abductive leap, lands the listeners in a new sphere of embodied knowledge, experienced prior to action, allowing them to inhabit the affective dimensions of the characters and the events. To Wenger, 'stories can transport our experience into the situations they relate and involve us in producing the meanings of those events as though we were participants'.[245] This results in the assimilation of the experiences acquired through narratives as the recipient's own lived experiences, and not merely something imagined. More specifically, in the case of a story map, the ability of the narrative about the yet non-existent aspects of the software system to be assimilated as lived experience tackles the complexity effects before the actual effects take place. At the moment of negotiation, the system's features may not yet have been created and otherwise only present for stakeholders as general and often vague descriptions. Yet, with the aid of story maps it becomes possible to introduce them into the field of negotiations, using visualisation as the tool for clarifying the narrative in all the necessary levels of detail.

In a practical sense, story maps are usually done on a wall with a series of sticky notes of different colours, bigger stories are negotiated and assembled in larger sequences as the knowledge around the system grows. The records are written down on sticky notes and arranged horizontally on a whiteboard or a wall to define the main steps. Each step is then broken down into constituent parts, which are positioned one below the other down from the main point. This creates a *map* of the body of a larger story, which serves as a description of the work that needs to be done. The constituent parts of story maps, the sticky notes called user stories, are the early drafts for the support tickets and have the same benefit of flexible movement and adjustment. The team can flexibly reorganise the stories and cut them together-apart, to use Barad's term, either by grouping the sticky notes, adding new ones as sub-tasks, or splitting some parts of work into later releases.

In the diagram of Fig. 15, the high-level user stories at the top are arranged left to right, pointing to the key steps of the particular user journey. For example, these could be the steps in the user's purchase on an e-commerce platform, in which case there will be steps such as searching for the item, adding the item to the basket, logging in to the user account, checking out, and so on. Each of the larger steps is broken into smaller sub-tasks, which are positioned under the main steps. Since the larger tasks can span multiple domains, the sub-tasks can be addressed to different teams, for example, searching for products in the shop catalogue may involve some tasks done by back-end engineers, and other tasks by sales and marketing teams. The attribution of smaller tasks to larger ones and their priority are easily identifiable due to their positioning underneath the user story. Where some steps require much more work than others, the delivery is usually split into releases, with higher priority tasks placed in the first

---

[245] Wenger, 2000: 203–204.

release. When defining priorities, the business owners negotiate with production teams to estimate the business value of each item.

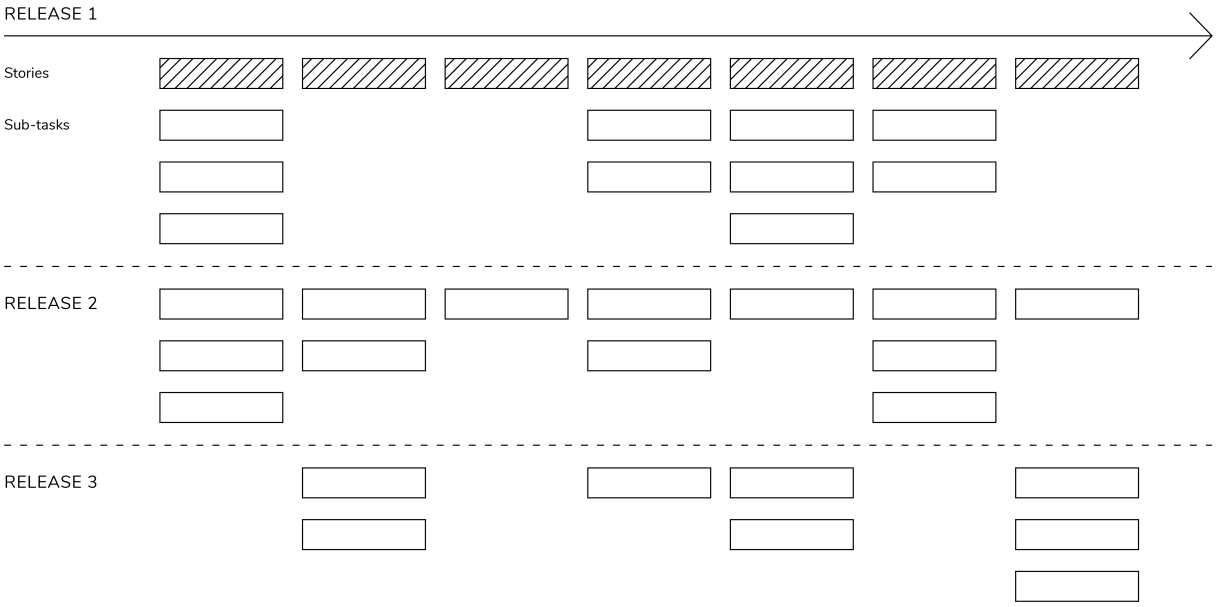

Fig. 15. The story map.

In its sense as a tool for model-based abductive reasoning, the story map as a type of diagram can be approached in terms theorised by C.S. Peirce as the visual stimulus, or a model, that helps the recipient to impose the order on the otherwise habitual and diffuse matters of perception.[246] By its quality of being present as an external referent, to Peirce, the diagram becomes a hypothesis which can be either accepted or submitted for further evaluation. Here it should be noted that the disruption-oriented workflow can, in fact, address the problem of accidental complexity raised by Frederick Brooks. As we saw, to Brooks, the software is too complex to be represented by a diagram because it consists of many moving parts that cannot be positioned on one plane for effective representation. The story map, however, achieves exactly that through making use of a few key attributes, which can be devised from the explanation provided by the philosopher of science Lorenzo Magnani on the role of visualisation in unveiling mathematical structures.

One such attribute is that the diagram enables an intuitive explanation of the concepts which are obscure, hidden, unjustified within the existing structure of knowledge, or otherwise too difficult to grasp within the capacity of the recipient's internal cognition. Furthermore, by enabling abductive manipulation, the diagram creates new concepts, where the constitutive parts or the strategy for organising

---

[246] Magnani, 2009: 35–36 citing Peirce.

them are yet unknown. Lastly, the diagram acts as an epistemic mediator and rather than bearing an explanatory function as is the case in its other two aspects, it reinforces the construction of what has been learnt and negotiated about the production system so far.[247] The use of the diagram for abductive modelling is therefore a reversal of what has been envisioned by Brooks. In Brooks's view, the diagram merely represents what is already there. In the abductive sense, the diagram is a creative way of engaging with the uncertain outcome, which can be constructed without exhaustive knowledge about the premises or methods.

The support ticket as a tool for the distributed practice of audit

The *practices of audit* specific to software capitalism are the self-observation techniques carried out throughout the production lifecycle by such means as version control tracking, code reviews and assessing documentation, requirements and inventory. The audit practices are understood from the standpoint of the general problematic of knowledge culture, in which knowledge acquisition presents a contradiction. On the one hand, if something is already known, no acquisition is required. On the other, as Karin Knorr-Cetina notes, if something is unknown, it is difficult to evaluate its quality, validity and adequacy.[248] Audit, thus, is seen as the epistemic practice aimed at addressing this problem, as well as the problem of review, which extends to assume that even if the knowledge itself is not new, it has either a new context or a new relationship with other previously known or unknown facts. For example, there may be new requirements about a certain component of software system that risk making it incompatible with existing component relations – in which case it requires a specific audit activity, referred to as regression analysis, to find out about it. Understanding audit as an epistemic event entails discussing it through and by comparison with the epistemic functions of observation, verification and validation.

A core attribute of audit is its strong sense of self-referentiality. The ability to self-reflect and comment is made possible through the implementation of standards. In its essence, audit can be defined as a practice of self-observation that occurs within a system in the shape of a formal 'loop' that evaluates the current data acquired in the field against the existing body of standards and the results of previous reviews.[249] Examples of audit activities in software production systems are the activities defined in Scrum methodology, among many others, as *retrospectives* and *backlog grooming*. The former is a group

---

[247] Magnani, 2009: 60.

[248] Knorr-Cetina, 2007: 367 with reference to Arrow.

[249] Lury, 2021: 69 citing Strathern.

discussion of work progress at the end of the significant delivery milestone, the latter is an activity of reviewing the issues that are not a part of the active rotation to either close them as no longer relevant or to schedule them for taking in. In both cases, the body of standards is present in the form of product requirements, business policies, and engineering best practices.

The contradictory situation is in that on the one hand, an organisation's operations necessarily create complexity, yet on the other hand, they have to continuously limit and reduce it. The former move is required because complexity is, as seen in the definitions provided earlier, something that comes as a by-product of the increasing density of communications between the growing number of agents and their assemblages within the system – such as among the company's employees, technical protocols or organisation's regulatory policies. Internal cross-communication is something that the organisation tends to promote because the increase in knowledge exchange leads to innovation, something that gives the organisation a competitive advantage. Nevertheless, such a move necessarily increases the complexity of both the organisation and, as a result, the production model it works with. In the other, reverse move, the organisation needs to pursue its goal of auditability, which implies that complexity is to be reduced, via team topology discussed in the previous chapter or scale-free architecture discussed in the next chapter, which would ensure its effective inspection. The section considers audit practices that counter the forces of complexity through non-synchronous task tracking that can be later gathered together in one place and reviewed. Once all the pieces are put together – in the general manager's office or on the Jira board or the Scrum master during the sprint review – it becomes possible to correlate sequences of actions with their results. In such a process of review, this chapter finds a novelty of the *new practices of audit.*

Defining tickets and tasks

Technical support activity has to be broken into the smallest possible parts that could be put down when there is no more time to work on them and picked up again when the next opportunity arises. This is the same argument for modularity that is used to optimise the organisation's performance, in the form Herbert Simon tells in his often-cited parable of the watchmakers. The first watchmaker assembles the watch out of individual pieces. The second one performs the task based on the ten assemblies, each consisting of ten subassemblies, each consisting of ten individual pieces. Every time the work of the first watchmaker is interrupted, she loses the progress done on the whole component and has to start on it all over again. When the second watchmaker has to interrupt, she only loses a small fraction of a subassembly.[250] Simon argues that modularity warrants the emergence of complex sys-

[250] Simon, 2019: 188.

tems out of simple ones, despite the absence of purpose that the watchmakers doubtlessly had. Economist and complexity scientist Brian Arthur draws the parallel between the parable and the division of labour principle of Adam Smith we saw earlier. Smith notes the benefits of splitting the workers into different specialisations, but only where the workload is sufficient. Arthur adds that the modularity of the watchmakers' division of work gives an advantage to the technological economy similar to the division of labour in manufacturing.[251] The advantage is the increase in utilisation of technology, and the ability of the workers to dedicate their undivided attention to a specific part of work, which lets them think creatively about better ways of carrying it out. Both features are crucial properties of the technical support workflow, where thorough partitioning means more work done in the long run, as it facilitates interrupting and resuming work as required without losing progress. Furthermore, the disruption-based workflow necessitated by the context of dysfunction as seen above, is established as the best practice of the support tickets, which can be found in one form or the other in most of the industry-grade project management software. The support ticket, in this case, appears as a perfect tool for handling the problem, which is not given but emerges through the sequences of actions. In its symbolic sense, the ticket stands for a specific aspect of a problem and therefore can be flexibly adjusted, split or merged with any number of other tickets, without losing the history of the interactions that it was a part of.

In the present context, a ticket, which is, in its essence, a written down work item, should rather be seen as the Agile tool for differentiation and integration of knowledge. In this capacity, the ticket links the attributes of a specific aspect of the production problem to the stakeholders. In one form or another, job tickets existed prior to Agile and in different contexts, such as lean manufacturing in Toyota's just-in-time production, where the tickets were arranged on the Kanban boards. Such boards are still in use in software project management. In the same tradition, project boards in Agile have columns which correspond to the stage of ticket completion, and the tickets are moved from one column to the next as the work progresses.

What makes the phenomenon of the ticket important to the present discussion is its central role in the integration process of the problem space of production. The reason for the ticket to play this role is because of its ability, as a record, to capture all the criteria that create the bonds between the problem, its problem space and the pertinent infrastructure of the knowledge around it. These are the criteria of requirements, which describe the business case, the acceptance which tells the stakeholders when the work can be considered finished, and the customer value, which instructs how to prioritise the ticket in

---

[251] Arthur, 2009: Ch.2.

the overall terrain of production problems. The three criteria that the ticket reflects have continuous relations with the specific types of stakeholders – one between requirements and production staff, the other between acceptance criteria and business owners, and lastly between the customer value and the users. The relations are achieved through making visible, tangible and manipulable, albeit quite explicitly made to change together with stories and conversations. Let me explain this by first giving a rationale for why the tickets should be seen as the best devices for such integrations, and then explaining each of their three aspects.

A ticket is a promise or a contract based on trust because it is present merely as a record of an intention or a request, not an order or law, in other words, it has an ideographic and not a nomothetic function. While existing within the problem space of the organisation's production system, it functions based on the reification principle of the community of practice, as the concrete manifestation of the abstract contradiction or error that in some cases of technical debt may not even be present in the production environment. While the work tasks associated with the ticket can be completed, leading to ticket closure, the ticket itself, due to the absence of the nomothetic constituent, does not have the capacity for being fulfilled. The requirements could be initially drafted on the ticket by the product manager, then be marked up, with items added or crossed out by the developers who do additional research, then comments would be added below, in the style of the blog or social media post, threading the discussion on any progress or further questions about the issue to other staff. Even after the ticket is finally closed, it is not erased from the system. Rather, it continues to exist as a record of a problem solution for any future reference, or even be available for reopening if the problem returns.

As the sociologist Michel Callon notes, such malleable and at the same time rigorously instructive characteristics of the support ticket, coming from its nature as a writing device, has a defining impact on the relations between the agents. The rewriting transforms the collective and individual participants as they 'participate in their own reconfiguration in the process of writing…. In rewriting, both collective and individual actors are reconfigured.'[252] The importance of making sure that all the participants know about what is being changed cannot be emphasised enough: in my fieldwork, I have come across cases when developers carried out the work without knowing that the requirements in the tickets were rewritten. After some frustration, we agreed as a team that any changes made to ticket descriptions after the work had started would have to be made clearly noticeable and marked in a different colour.

---

[252] Callon in Law and Mol, 2002: 204.

Using support tickets in production work

To describe the practical application of production software for tracing the circuit of epistemology and the problem space of production, it would be useful to briefly describe the way I've been using the tools in my day job as a product lead, where the software has allowed me to situate the work within the four parameters of the problem space. On the one hand, it provided all the possible variants of arrangement of job tickets that could be activated in different combinations to describe a specific problem. On the other hand, backlog administration had ensured a solid grounding in the form of a mature epistemic infrastructure that made problem space easy to navigate. Where the previous thesis chapter saw the infrastructure deployed as code every time the system is called, the problem space can now be seen as equally flexible, composed based on the problem at hand. The important prerequisite is that each problem, whether a code bug, a feature or another piece of work, has to be anchored via the three categories of the problem space of production: to be described in the requirements, to have a definition of done and to be evaluated in terms of business value and the story point estimation. Once the problem is established as such, it gets included in the mapping of the problem space, which is then carried out based on these criteria in the manner of the existing production workflow. In my practice, after some trial and error with various combinations of software tools, I have come to use the Atlassian Jira bug-tracking system in conjunction with Smartsheet for project Gantt. Jira allowed me to create a card-based catalogue of records, each containing a specific issue, and to sort, link and search through them. Smartsheet is a spreadsheet software that became a useful complement to Jira because of its ability to create Gantt charts, a distinct visual device I briefly introduced in Chapter 1, which is used by project managers for time planning. Such charts are unique in that they combine the project's tasks, the task descriptions, the staff member doing it and the timeline for their completion – all within one graphic representation. This enables working with either all these properties simultaneously or creating more granular reports around specific attributes of a project or a stream. When used together, the two tools afforded a thorough insight into the problem space, specifically due to their sophisticated mechanism for searching and filtering.

An important aspect that ties together this chapter's discussion of cognitive load and the cognitive shock described in the next chapter is the support ticket property that prohibits the ticket to be assigned to more than one person at any one time. This serves two purposes: on the one hand, it traces the outer limit of each specific problem that the person involved with it needs to tackle, which helps avoid the shock associated with excessive integration. On the other hand, it helps to avoid the confusion of deciding authority, which in the cognitive sense is a problem of communication throughput: if the ticket is assigned to more than one person, then neither of the assignees are going to do it because none of them would be able to know, without further inquiries, whether another person is already

working on the ticket. No additional communication load is added if the assignee is kept to one worker.

In the closing of the discussion on tickets and the role they play in audit, it is necessary to emphasise that despite all the benefits, support tickets have the same shortcoming as the other audit practices, in that they cannot account for tacit and transindividual aspects of work. The distinction is explained by Star as the production aspects of the work as opposed to articulation aspects. She contends that one is unthinkable without the other. Production work is everything that is written in the tickets and expressed via the criteria of the problem space – acceptance, requirements and customer value. The flip side is all the work that binds the criteria together into the assemblage, the complex entanglement of organisational routines, strategy and motivation that makes the stakeholders participate in the negotiations, to begin with. Only by taking into account both the explicit tasks of production and the hidden work of articulation, it is possible to 'come up with a good analysis of why some systems work and others do not.'[253] The motives of the agents, as the earlier discussion of agent behaviour in complex systems in Chapter 1 has demonstrated, differ in top-down organisations and self-organising agent cooperatives. Pertinently to the present case, it could be concluded that there is another case where the community of practice dynamic seeps through into the sphere of the organisation's operation. This is because the agents will not be motivated merely by the organisation's goals and clearing of milestones in the project's Gantt, and, as Azadegan and Dooley remind us, seeking instead validation through the alignment of their skills or reputation with the tasks they carry out.[254]

Chapter conclusion

This chapter has proposed the DevOps agenda of iterative development of an organisation and a software production system in response to the complexity challenges which are introduced by their mutual co-implication. The stress relation, on the one hand, discourages the organisation from adding too many new staff members at once, and on the other hand precludes the state of correct functioning of the system, replacing it instead with a minimal deployable state, which seeks to cut the spending of resources on system's repair to the bare minimum. There is an organisational tendency to absorb the complexity effects by dispersing them into the distributed collectivities, such as communities of practice (CoP) therefore displacing its function from addressing the dysfunctions to managing their continuities. Besides absorption of complexity shocks, however, CoP also tends to solidify in its opinions and methods, often acting to stifle the individual initiatives of its members and inability to rapidly response to

---

[253] Star, 1999: 387.

[254] Azadegan and Dooley in Allen et al., 2011: 426.

change. The consequences of such momentum in distributed systems are not as harmful as in centrally-organised hierarchies because the decisions do not require validation of any central authority and are tried and tested locally. This enables a more organic response mechanism to the external stimuli, in relation to the learning capacities of the individual agents – something which the next chapter discusses in more detail.

Furthermore, the chapter has demonstrated that the software production system's distributed principle in combination with its requisite dysfunction shifts the organisation's main concern from production to managing dependencies in such a way as to minimise the repair costs while meeting most of the problem space criteria. The workflow in this sense is better explained as disruption-oriented, that is, organised in such a way that permits stopping and re-starting at any moment. The chapter has described the support ticket as the central vehicle of such workflow diffractively, or in terms of the multiplicity of its appearances. It acts as an abstract tool for the symbolic manipulation of work, providing an ability to split and re-organise any aspect of a problem to create a corresponding complexity of its epistemological infrastructure. That is, the problem description can grow from one to numerous tickets in case its complexity spikes, and yet just as easily can go back to one ticket again if the complexity reduces, or if its aspects take independent trajectories within the problem space. Furthermore, the support ticket is concrete enough to be present as a contract that binds together all the parties involved with the problem by creating evidence. This is made possible by the fact that each ticket adheres to the governance protocol, discussed further in the next chapter, which enables it to be audited in a scale-free way.

So far, the discussion has approached the complexity avoidance in the software production lifecycle model in terms of its more analytical tactics, which aim at splitting the larger problems into smaller parts, either in the deployment pipeline or in the problem space negotiations. It is now necessary to turn to the opposite move of integration, which brings the smaller parts together into a larger infrastructural whole. The problematics of integration shed light on the complexity effects that prevent large technological systems from responding to change rapidly and examine the internal systemic mechanisms that allow them to realise the distributed practices of audit. In other words, I ask, how is it possible that under such severe complexity conditions as described up to now, any software production is possible at all.

Chapter 5. The governance and the falling cost of computation

The systems theory approach, as described in this study up to now, views an organisation through its temporal and spatial patterns of operation. In terms of temporal flow, the organisation generally processes the inputs, such as resources and customer orders, into the outputs, such as products and services. In terms of space, the organisation establishes itself topologically by splitting its internal protocols from the ones of the external environment, as Mezzadra and Neilson observe, imposing the boundary as the tool for legal ordering and enactment of market relations, and acts to maintain the equilibrium between the two so that the boundary remains intact.[255] It may be viable to suggest *governance* here as an organisational practice that makes the conversion of inputs to outputs possible by regulating the split between the organisation's inside and outside. Along the same split, there are corporate and business kinds of governance. The former deals with matters such as policy compliance and consistency with the audit practice, while the latter is concerned with business performance and the creation of value. The practical job of governance, to use David Farley's formulation, is largely the responsibility of the boards and executive staff which have the authority to provide the strategic direction and verify that the organisation's resources are used responsibly.[256] In this chapter, I am concerned with the changes that governance needs to undergo in the face of the extreme complexity of things, the production of which it has to regulate – a discussion, which is appropriate for the final chapter of the thesis because its matters relate my research to the wider body of the digital humanities and cultural studies more generally.

The chapter revolves around three main themes: the falling cost of computation, the problematics of assimilation of knowledge and the coordination specificity in distributed governance, which I approach by splitting the discussion into three sections. In the first section, I introduce the trend of the falling cost of computation by establishing what computation means for production, why its costs tend to diminish, and what kind of benefits and risks the trend presents. I find that the falling cost of computation arises from the exponential increase in the computational capacity of hardware and tends to escalate the complexity of the production system as the software capitalist tendency seeks to valorise the

---

[255] Mezzadra and Neilson, 2019: 215.

[256] Humble and Farley, 2010: 417.

computational surplus. The increasing computational capacity, as the second section discovers, has a direct impact on the system assimilation of knowledge through increasing the agent's cognitive load, or the amount of mental effort they have to exert to support the production flow. In the situation where the complexity pressure cannot be alleviated, the agents treat the incoming knowledge parsimoniously, by codifying and abstracting it – that is, by applying the categorisation and other patterning rules to the knowledge so that it becomes easier to navigate. The more successful rules and patterns tend to propagate throughout the system and evolve to become operative on the levels of teams and organisations. In the third section, I treat the parsimony principle as the evidence of self-organisation that makes the governance distributed, and concomitantly as the key effect that complexity has on administrations. The advantage of self-organisation is that it is more resilient in the face of change, as the new methods get continuously promoted from the agent level upwards in case they appear to be more adapted to the changes of the environment. The high adaptability explains why the systems that realise the distributed governance model are often referred to in complexity management theory as *complex adaptive systems* – their complexity refers to self-organisation, and adaptability points to their resilience. The aim of presenting the organisation as a complex adaptive system is to verify that such presentation makes it possible to think strategically about complexity avoidance in the context of the falling cost of computation.

Computation in production

The ubiquitous use of computation in all the processes of software capitalism confronts the present study with a necessity to account, however briefly, for the consequences that computation's key tendency – the disruptive diminishing of its cost – has on the production lifecycle and the complexity forces therein. To summarise what we have learnt about computation so far, it makes sense to revisit its definition adopted for the present research, and the benefits traditionally ascribed to it, prior to delving into a more detailed discussion of the associated risks. I define computation in terms of Brian Arthur's argument of the counterposition between algebra and computation in economic theory. To Arthur, economics had historically used algebraic statements and thus had evolved to be described with nouns, becoming largely equipped to think quantitatively, which supports exact and reliable explanations.[257] *Computation* is a process of executing a set of instructions or operations, or an algorithm, within a system. Unlike algebra, it lacks the certainty of interpretations but instead enables a focus on the domain under investigation in terms of its processes.

---

[257] Arthur, 2021: 9.

The key shift here is from a method that uses nouns to one that uses verbs, which is also present in compositional methodology. As Lury notes, the present form of the verb in the description of method emphasises the involvement of the event of the application of method into the matters of research, and therefore activates the situation as a problem.[258] And, as Arthur contends, the use of verbs can offer a fuller description of complex systems through the heterogeneity of agent positions, through shifting emphasis from objects to actions and describing the models in uncertain circumstances not only with statements but also with processes and flows they are involved in. Computation operates with an *if–then* boolean logic and is, therefore, something that is far more useful for negotiations in the problem spaces of software production systems, which run simultaneously through the organisation's lines of communication, its deployment pipeline and its business value stream.

The benefits of depreciating computation

The professional DevOps literature the previous chapters looked at follows a path of a predominantly technical discussion of software that focuses on the advantages of the expected ongoing increase in the computational capacity of the hardware. It is usually implied that the existing issues of negotiations within the problem space of production will be easier to solve as the drop in the costs of computation continues, along with the associated upgrades to the production equipment and the new conceptual approaches to production, such as Continuous Deployment and team topology, discussed in previous chapters. The benefits thus offered can be split into two categories, occurring either on the individual or on the group level. On the individual level, there are *data processing* payoffs, which see the increase in efficiency of agents, whether human, non-human, organisational, or other types. Second, there are *data transmission* benefits, which look at the improvements in exchanges between the agents. In the case of hardware microprocessors as discussed by Intel's Gordon Moore, the two categories are mutually reinforcing because, as Max Boisot notes, the higher speed of processing also means higher throughput in transmission.[259]

Furthermore, the higher rates in data processing and transmission result in corresponding increases in knowledge diffusion and a higher bandwidth effect. On the one hand, the knowledge diffusion ratio indicates how easy it is to access knowledge: the lower the cost of computation, the larger the number of people to whom this knowledge is available. The high rate of diffusion may mean that the epistemic infrastructure can be made accessible to all relevant members of the production event, subject to user management policy. This, in turn, makes viable the continuously integrated production lifecycle model

---

[258] Lury, 2021: 17.

[259] Boisot and Li in Boisot et al., 2007: 129.

as simultaneously the carrier of the business value and the deployments of software stock produced. On the other hand, the bandwidth effect, as theorised by Max Boisot, implies a more dense and rich character of communication in general and digital media distribution in particular.[260] Where previously the communication required more effort on the side of stakeholders and was constrained to specific formats, such as voice-only calls or print-outs of visual materials, the higher bandwidth allowed negotiations using face-to-face video calls and screen sharing. Additionally, a higher bandwidth meant that the problems that could previously only be solved by the means of non-digital delivery could now be addressed via computation-based means. For example, the published media initially required the physical distribution of printed materials to the audience. The increase in bandwidth has enabled a gradual shift in solving the problem to the realm of the digital – starting with the use of electronic communication and digital printing, then by switching to fully digital online production and distribution without any printing at all.

The traction crisis diagram from Chapter 1 (Fig. 1) can be revisited in this context to illustrate how complexity appears in situations where processing and transmission are rising, depending on how the organisation follows this up (Fig. 16). In the first instance, the governance stays fully or partially centralised, which means retaining some of the complexity effects because of the inability to keep up with the increasing pressure to audit the increased knowledge traffic. The other choice is to go distributed, which balances the complexity out through scalability of control, such as through designing the problem space architectures to be self-similar throughout the various abstraction levels.

---

[260] Boisot in Boisot et al., 2007: 160.

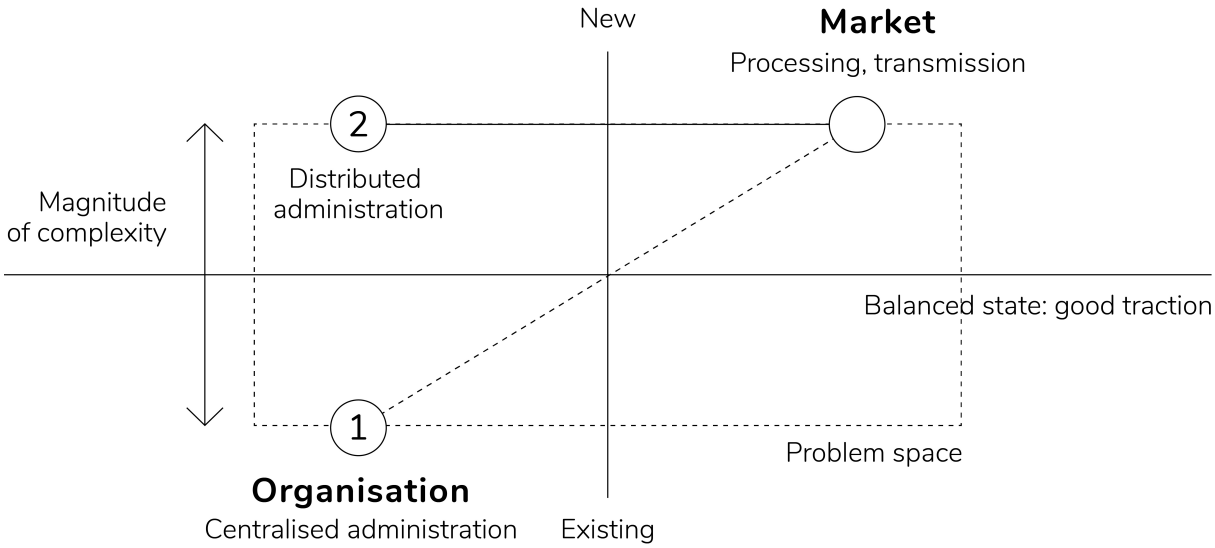

Fig. 16. Traction balance using distributed administration.

The risks of depreciating computation

Despite such benefits of the tendency of the cost of computation to fall, the interest of this chapter lies in going in the opposite direction and outlining any disadvantages and risks. Is it possible that low-cost computation introduces complexity that could have otherwise been avoided? Keeping in mind Melvin Conway's observation that complexity in a software system tends to be reflected in the organisation's communications, locating the failures and understanding their causes should help shed light on the risks cheap computation introduces to the organisation and market domains. Some of the more prominent disruptive effects of the cheapening of computation that this study looks at are the software crises. It should be noted that while the software crisis as a term is equally applicable to all the crises which have occurred within the software production practices, this in no way implies that the substance of the crisis itself has never changed. Quite on the contrary, this is precisely because the crisis always emerged in a different shape, which made it cause enough distress to be regarded as a crisis all over again. The examples this thesis looks at are telling yet by no means aimed at providing an exhaustive list. In the 'big iron' era of mainframe computers such as the ones described by Frederick Brooks, the software crisis was manifest in the organisational reverse salients that led to staff increases and clogging of communication channels, ultimately causing the organisation to stumble in the allegorical tar pit. At a later stage, once Agile thinking started becoming widespread, the software crisis shifted to signify the impossibility of integration. There was no longer a shortage of code libraries and other components, and the main stumbling block has become the maintaining of parts, keeping them compatible and tracking the changes. This has become known as technical debt, similar both to real financial debt and to the previous software crisis, threatening organisations to go bankrupt due to the inability to maintain their own code. Once the disciplines of code reviews and refactoring were estab-

lished, it became possible for the organisation once again to position its efforts orthogonally to the complexity forces and to avoid disintegration under their unbearable burden.

However, neither the Agile workflow nor the code review practice stopped the complexity effects from emerging in a different form. As Chapter 1 aimed to explain, the systemic approach to production comes together with a centralised model of governance enacted via the intra-organisational management structure. As Chapter 4 discussed further, such structure is necessary for epistemological and economical facilitation of audit. In other words, since the audit is only possible of something auditable, the internal system's management is implemented in such a way as to provide the organisational setting in which the audit is possible. Review no longer has to deal with the full complexity of software production systems, but only with the management tasked with reproducing the organisational structure all the way down with the auditability in mind. While being effective enough in dealing with auditable matters, the downside of such an approach is that anything that falls outside of the audit capabilities also cannot be accounted for when negotiating problems, and therefore is not a part of the problem space of production. Meanwhile, there can never be a shortage of new factors of production that cannot be accounted for, brought about by the new and more capacious hardware, in alignment with Moore's Law. This is, the chapter argues, where the further disruptions and software crises come from. With this in mind, the chapter aims to analyse the disruptions along the axes established before as the conditions for system cohesion – change, momentum and control.

The tendency of the cost of computation to fall is a reversal of Moore's Law which puts an emphasis on the value created by the hardware, as a tool of production, in relation to the cost of production of said hardware. In other words, since the computational capacity of microchips is growing, it becomes cheaper to produce a microchip with the same computational power, and the amount of computation that in the era of 1950s mainframes required large-scale investments into operations and infrastructure is now possible with the relatively low-cost solutions.[261] To understand why the falling cost of computation is a risk, it is best to turn to the early problematisation of it by a pioneer computer scientist, Douglas Engelbart. Widely credited as the inventor of the computer interface, throughout his career, Engelbart was broadly engaged in developing a comprehensive framework that would tie together social and technical aspects of personal computing.

---

[261] As the roboticist Rodney Brooks observes, 'a week of computing time on a modern laptop would take longer than the age of the universe on the 7090,' (Brooks, 2021) – which effectively suggests that the computation has lost at least that much of its cost.

In the 1994 interview, Engelbart had warned that the real social danger of technology is its disruptive expansion: 'the rapidity with which really dramatic scale changes are occurring in what the capabilities of technology are, are such that by the time that really gets integrated into the whole, our whole social human system there's a lot of adaptation to be made'[262] In the current industry sources, this concern is echoed by the business analyst Azeem Azhar, who argues that businesses that consider the falling computation costs are better positioned to take advantage of its effects – with the primary benefit of not being crushed under the mounting complexity of production. 'One primary input for a company is its ability to process information. One of the main costs to processing that data is computation. And the cost of computation didn't rise each year, it declined rapidly…'[263] For software capitalism, processing of information is the key function of any business. The firms, Azhar observes, 'are largely not cut out to develop at an exponential pace, and in the face of rapid societal change.'[264] Yet, as this study saw up to this point, slow adaptation is, in fact, a part of an organisation's survival tactics, since it promotes the organisation's healthy traction, or deployment-to-integration ratio between its EIAC and the problem space of production. Good traction lets the business model generate sufficient outputs while being able to manage the inevitable incoming changes and stay internally coherent. The balancing of the three aspects of traction – momentum, change and control, becomes increasingly challenging in the face of the exponential fall of computation costs. The key challenge, as the next section uncovers, comes from the cognition processes associated with the ways in which the agents treat the incoming flow of new knowledge.

Data, information and assimilation of knowledge

To unpack the specificity of the assimilation of knowledge in its relation to complexity, it now is useful to examine the principle of requisite variety, developed in early cybernetics and frequently employed in organisation studies to describe the behaviour of agents in complex systems. Formulated by the pioneer cybernetician Ross Ashby, the so-called Ashby's Law of requisite variety states that to survive, the system needs to be able to develop a response mechanism which is capable of countering the full spectrum of stimuli coming from its environment – save for the noise.[265] If the system does not produce enough variety, it fails to secure the resources it needs to survive and eventually fossilises. If, on the contrary, it overreacts and spends excessive resources to produce more responses than the external stimuli require, it risks disintegration. Once the system achieves the point of requisite variety, it is selective

---

[262] Engelbart, 1994.

[263] Azhar, 2021: 85.

[264] Ibid.: 101.

[265] Boisot and MacMillan in Boisot et al., 2007: 51.

enough and achieves sufficient parsimony to be able to allow for a response relevant to its environment. What such a presentation reveals about the software production system, is that the interactions of agents are necessarily mediated via the problem space of production and that in fact, two integration processes take place – on the one hand, the knowledge is assimilated by the agents from the problem space, and on the other hand, the results of the problem negotiations are integrated into the EIAC. Such integrative processes allow the system to maintain the complexity requisite to its purpose.

In the agent-level integration (Fig. 17) developed by the complexity management scholar Max Boisot, the agent acquires the stimuli inputs from the problem space of production and uses a range of perceptual filters informed by the context of the organisational culture and the agent's own expertise to recognise the data from noise.[266] In the next step, the agent's conceptual filters are applied to detect the patterns in the acquired data to filter the relevant items, which collectively become the information. Lastly, the information gets assimilated as knowledge via the agent's personal considerations based on its stored mental models and organisation-specific values that have pre-existing relations with the information inputs. Even though the three types of production inputs – data, information and knowledge – may seem like the various stages of gradation from signal to noise, in fact, the distinction is context-specific: what is noise in one moment of an agent's cognition can be a relevant piece of information in a different context.

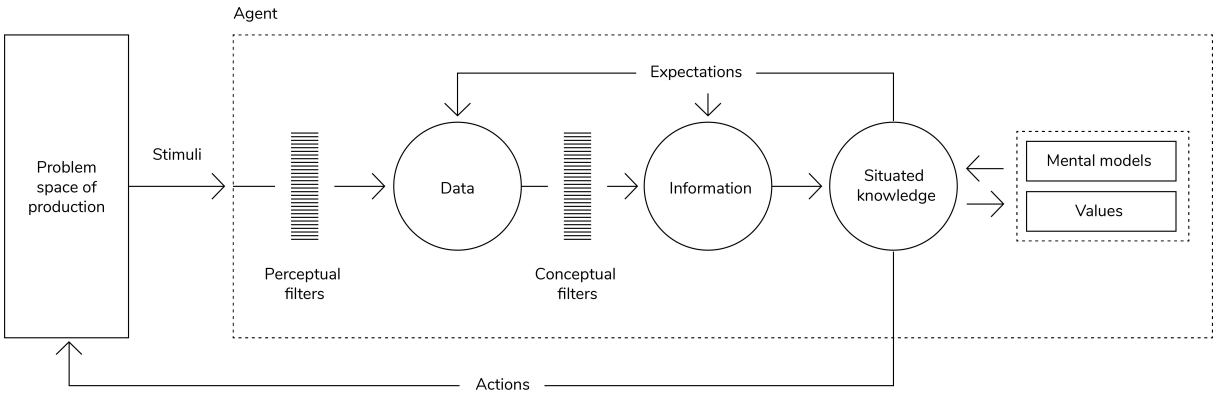

Fig. 17. Integration as the process of assimilation of knowledge inventory by the agent. Adapted from: Boisot and Canals in Boisot in Boisot et al., 2007: 20.

The inputs therefore are best seen in terms of their utility and the impact they bear on the agent's behaviour. The utility is different for each of the three inputs. The advantage of data is that it can carry any sort of evidence without discrimination. The benefit of information is that it can inform the agent

[266] Boisot and Canals in Boisot in Boisot et al., 2007: 20.

about what can be expected in terms of knowledge it may bring. The utility of knowledge itself is in its ability to modify the agent's actions in a way that makes it better adapt to the environment. As Boisot finds out, knowledge is the set of beliefs held by an intelligent agent that can be modified by the reception of new information and on which the agent is disposed to act.[267] It should be noted that in the problem space, proximity is related to such behaviour modification. While Fig. 17 portrays the relation between the individual agent and the whole of the problem space in an abstract way, the problem space itself is not homogenous and the agents are more likely to receive most stimuli from the agents located close to them, albeit through the mediation of the problem space.

As a second consideration, besides the integration process from the problem space by the agent, there exists the integration that assimilates the negotiated knowledge from the problem space of production to the EIAC. Such integration is widely discussed in professional DevOps literature under the general rubric of Continuous Integration. In a move opposite to the Continuous Deployment discussed earlier, Continuous Integration is seen as the operations' best practice that requires the system code to be frequently integrated into a shared repository. The difference between continuous approaches to deployment and integration, for the present discussion, is that Continuous Deployment makes the system available for the stakeholders, while Continuous Integration deals with the maintenance of infrastructure and is aimed at making sure that EIAC is always up to speed with all the recent changes. In the practical sense, Continuous Integration means that the whole of the system's code is getting synchronised once every few minutes to eliminate the risk of failure when merging big batches of code which might not be compatible and carry the potential of breaking the system. Discussion of the epistemic infrastructure in Chapter 3 may lead to thinking that once the infrastructure for knowledge is in place, and the system for capturing and accessing the knowledge is established, the actual process of knowledge acquisition becomes a rather practical consideration. However, what organisations quickly realise is that not only the infrastructure needs to be constantly realigned with the knowledge contained in the organisation, as this thesis pointed out before, but the knowledge itself is not something that can in fact be recorded into the provided framework. To understand this problem, some consideration has to be given to the differences between knowledge and information, as it is discussed in Brown and Duguid.[268] First, information is more independent and resides outside of the space of action, that is, has an infrastructural quality. Knowledge is more active, it needs to be associated with the knower, the practitioner or other sort of agent – in the case of the problem space of production, they are stakeholders, including users, business owners and production staff. Second, information, as more self-con-

---

[267] Boisot and Li in Boisot et al., 2007: 117.

[268] Brown and Duguid, 2000: 119.

tained, is easier to transfer. It can be written down in code, placed into the database, and can be referenced. Knowledge, on the contrary, is something that cannot be pointed to directly – instead, the referral needs to be to the person who knows a particular thing. This makes the whole discussion about access rights and permissions more complicated. Third, knowledge is something that requires assimilation. For example, when a specific record is accessed in the company's support wiki page by a member of staff who accesses it to decide on a given case, it needs to become knowledge through the process of understanding – to be achieved, knowledge requires a certain degree of commitment. For example, 'I have information, but I don't understand it' is possible, while 'I know, but I don't understand' is not.[269]

Consideration of these properties of knowledge helps to clarify why epistemic infrastructure is not sufficient on its own, as the repository of software system possibilities. Once organised and deployed, it still remains, albeit informative, rigorously structured and topologically oriented. On the other hand, the problem space, as the activities within it evolve, becomes increasingly dependent on infrastructure. The organisation in software capitalism is primarily a learning initiative, which means that it depends for its operations on human and intellectual capital circulation, manifest in the acquisition of knowledge.[270] Therefore, it is imperative for it to see the assimilation of knowledge, or its *integration*, as the primary function. In other words, if the deployment function is the essence of the EIAC, then the integration function is the essence of the problem space of production. Simultaneously, the software capitalist formation at stake here, which is also frequently referred to as knowledgeable or cognitive, makes it quite prominent to the businesses that an employee's mental capacity is a valuable company asset, and that the knowledge is contained in the minds of the teams, rather than in the databases, since, as we saw, knowledge is not something that exists outside of the knowing subject. Departing from the industrial mode of production that only viewed living labour as adding value in its capacity as a machine operator and its maintenance, in the new form of production the value depends on the human ability to learn. Two noticeable moves make the treatment of knowledge more explicit in IT production. On the one hand, there is a practice of making the information relevant and easily accessible, which for the present context is referred to as differentiation. On the other hand, there is the explicit incorporation of the practices of finding, organising and presenting information into the problem space of production, which requires establishing a more thorough understanding of the process of integration. To zoom in on the specificities of the differentiation and integration, it is necessary to start from a wider discussion of the cognitive load metric, which both events could be accessed with.

---

[269] Brown and Duguid, 2000: 120.

[270] Lury, 2021: 180.

Cognitive load and governance of the knowable

The topological becoming of problems in problem spaces not only clarifies their mapping to epistemic infrastructure but also activates the mechanisms of governance over access to what is known. The governance regime implies a specific set of values based on the capacity to know, establishing a protocol for knowability that operates within a specific imaginary. As Lury puts it, 'problems always become topologically, with-in and out-with problem spaces' and there is 'an imaginary of knowability, in which it seems everywhere there is a capacity to know.'[271] Thus, the imaginary sets up the common beliefs about the construction of knowledge and the protocols of access. The problem with the implementation of such a control mechanism in the context of the radical collapse of the cost of computation, however, is that any attempt is thwarted by information overload. This makes it important to consider the limit to how much knowledge stakeholders are capable of acquiring and processing as they get practically involved with negotiating the problems within the problem space of production – the *cognitive load*.

Cognitive load is associated with the total amount of mental effort exerted by the individual's short-term memory while engaging in a problem-solving activity. As discussed in Chapter 2, the basic attributes of such activity are present in the problem space as the givens, the goals and the operators. During problem-solving, as psychologist John Sweller observes, an individual creates an inferential mesh to capture the available operators that promise coherence between givens and goals. The process demands that the individual simultaneously considers many aspects of the process, such as the current problem state, the goal state, the relation between the problem and the goal, the relation between any operators involved, and if the problem consists of many sub-goals, any updates due to the goal stack as the problem-solving goes along.[272] Having to keep all this information in the short-term memory at the same time increases cognitive load. Furthermore, problem-solving can involve additional steps. For example, if the goal is unknown, the problem solver needs to make a series of abductive leaps into the unknown, a process that involves an intense period of learning about the essential structural characteristics of the problem space prior to any specific goal allocations. For this reason, while being creatively stimulating, abduction can be a taller order on the solver's attention than regular means-ends inferences.[273]

---

[271] Lury, 2021: 184 citing Thrift and Hayles.

[272] Sweller, 1988: 261.

[273] Ibid.: 260.

Some of the factors that increase the complexity of the problem space of production are, as Callon observes, of the computational nature. They can involve software systems in the form of domain conflicts or the institution itself, in terms of its social dependencies. The former introduces new, disconnected bodies of knowledge, while the latter risks disrupting existing dependencies or creating new ones, the result for the agents in both cases being an increase in computational load associated with building new connections. Some examples of risk factors are tightening of competitive constraints, escalation of changes in production outputs or services that would demand faster adaptation or the re-organisation that brings new teams and specialisms that increase the heterogeneity of workflow processes or inventory.[274] Speaking more generally, the criteria of the problem space, which are split for the present study into requirements, acceptance and customer value, can be more varied and specific to division or initiative. Whichever the factors are, the change they introduce is complex because it moves simultaneously along the two axes. It is also computational because it informs the procedural pattern of agent behaviour. On the one hand, the change increases the number of variables that have to be accounted for to understand the behaviour of the system. On the other hand, it intensifies their nonlinear interactions.[275]

In DevOps, cognitive load balancing is seen as the risk mitigation tactic which helps to avoid situations where the excessive amount of responsibility makes the work of the team perforated as they switch between increasing amounts of concurrent processes. It deals with the notion of the domain, which implies that the team members are located within a specific set of situated knowledge, which helps to prevent crisis events associated with the acquisition and processing of new knowledge, such as cognitive shocks. A *cognitive shock* should be defined in light of the present thesis' view of complexity as a production failure caused by the disruption in either of the two aspects of routine cognition of the production system circuit. On the one hand, there is a limit of communication throughput where multiple individuals are involved, and on the other hand, there is a parallelism of individual activities, when cognition is carried out within one mind. The latter aspect poses a limit to how much knowledge integration one human mind can carry out at any time. To draw an example from Sweller, since the human short-term memory is limited, the cognitive load increases whenever problem-solving requires storing a large number of items within the short-term memory.[276] This limit is higher when the system consists of many minds, however when multiple individuals are involved in problem-solving activity, there is a limit on the communication bandwidth among the independent minds that cognition is distributed.

---

[274] Callon in Law and Mol, 2002: 191.

[275] Boisot in Boisot et al., 2007: 155.

[276] Sweller, 1988: 265.

Cognitive shocks occur when the agents have to assimilate excessive knowledge inventory, such as when switching between the domains, as indicated above. The disruption occurs whenever the knowledge about the inputs that the agents need to transform the incoming data into the outputs do not have any familiar or regular patterns. Patterns may be lacking, according to Matthew Skelton, because the spheres of knowability between the domains vary, particularly in the production of such complex artefacts as software systems, which may deal with numerous domains containing little or no compatible knowledge patterns that the agents can use for differentiation and integration.[277] Skelton recognises intrinsic, extraneous and germane types of cognitive load agents usually deal with in software production. *Intrinsic* load deals with the acquisition and processing within the immediate vicinity of the agent's location in the problem space: libraries, languages, classes or plugins. *Extraneous* load comes from environmental pressures, for example in component deployments or integration of components into the system, or use of testing suites. *Germane* load pertains to the tasks which require new learning outside of the agent's immediate area of familiarity, for example, anything related to the business domain that the components are delivered into, such as cross-service compatibility considerations.[278] Load balancing splits the different types of load to the duties of different teams, that either align with the technology stream or with the domain. Intrinsic load deals with the internal issues of the domain. It has the least impact on the overall topology of the problem space since all of the complexity is encapsulated within it. The other two are related to cross-domain processes, which means that the boundaries have to be carefully navigated to avoid cognitive shocks.

The key challenge in the situation of escalating change is that there is a demand for agents to constantly switch between the different kinds of cognitive load, from intrinsic to extrinsic to germane, and simultaneously update the whole knowledge stack to bring it back in sync with the new standards or frameworks associated with the changed processing and transmission capacities. The change in context requires re-indexing of the production system's components to address the evolved set of external stimuli, as per Ashby's Law of requisite variety and mapping it to its existing EIAC – the appropriate codification and abstraction processes. Such was the case in my empirical study at JX, the online publishing platform mentioned earlier. Following a critical security incident, it was decided to carry out the system migration process to AWS public cloud. The migration meant an improvement in the system's traction because it provided better control over its resources and allowed it to deploy frequently, thus maintaining a good rate of change. It, however, also meant a substantial upfront investment to be able to ensure that new knowledge is compatible enough with the existing epistemology. This had occurred

---

[277] Skelton and Pais, 2019: Ch.3.

[278] Ibid.

in the JX production team during the switch to AWS, albeit with no considerable impact on the delivery, since the appropriate onboarding procedures were in place. Still, in the aftermath of the migration, the production routines had to be re-learnt because of the introduction of new tools such as Jenkins for the deployment pipeline management, and new approaches to organising branches and releases in GitHub. It could be predicted that in cases where the change is more disruptive than the adoption of a new cloud service, it may not be possible to avoid, resist or predict the cross-domain cognitive switches within teams, which may lead to escalating disruptions in delivery.

Beyond the complexity caused by the frictions between the domains, another aspect of the cognitive pressure that the knowability governance needs to account for is the strong association between computational and social dependencies within the problem space of production. Since knowledge is something that has to be assimilated in an explicit event of learning, any discontinuities in social interactions may lead to spikes in cognitive load, which makes any collective endeavour within the organisations a computational as well as social occurrence. Such are the negotiations between stakeholders, which can be considered computational events in that they follow specific protocols or routines. Any computational event presupposes communication between the team members. More specifically, if one part of the computation is the responsibility of one agent, another part can be the responsibility of a different agent, depending on how the knowledge is diffused. Something that begins as a design job can later become a development job, and later yet, an integration or testing job. Regardless of team topology, teams and domains are closely interlinked, computationally as well as socially. As cognitive scholar Edwin Hutchins observes, competent load balancing is linked to the effective managing of dependencies because each part of any problem negotiated is not only a computational event, but is simultaneously a social message.[279]

This means that the resilience to complexity within the organisations, including the ones that employ topological strategies to design their production lifecycle, depends, to a large extent, on the cohesion of the social structure. The reliance on social dependencies can be so strong that it makes Hutchins question whether it is more valid to say that car production is the primary outcome of the labour of the car factory workers as a company department. For all they know, the primary role of the organisational form could be to produce a specific social dependency schema, with cars being an additional benefit, more relevant to the model of the market involvement of the business, rather than to the organisation itself.[280] This notion is also reflected in the value-based market relations, which, as Marx develops in

---

[279] Hutchins, 2000: 224.

[280] Hutchins, 2000: 225.

Volume One of *Capital*, result not only in the output of commodities but in further reinforcement of the classes of the capitalist and the wage-labourer. Marx observes that the capital produces 'not only surplus-value, but it also produces and reproduces the capital-relation itself'[281] In other words, since the capitalist mode of production cannot be interrupted in the interest of a continuing generation of surplus value, the production of commodities always includes the production of the social misbalance that makes it necessary to continue the production of commodities. Reproduction of social relations of production in a software production system falls squarely in the purview of DevOps. This has to be the case due to the requirement of auditability, which means that any relations that are not recorded in the form of EIAC, and are not deployed and integrated continuously, are vulnerable to the complexity effects. This makes it necessary to address the auditability consideration of scalability.

Fractality and scale-free systems

According to the thesis' earlier observation, whenever changes are introduced into the system, the system tends to respond with the complexity increase. Such an increase presents a problem for planning and review, bringing uncertainty into the audit practices, which in turn threatens to halt production efforts. This means that keeping things auditable is a requirement which can be resolved by applying the additional work aimed at continuously simplifying the system. Such a method, however, may appear unsustainable in the face of exponential change, since, together with the expansion of the system, it would require equal rapid expansion of review and refactoring efforts across the many of the system's components that suffer the complexity effects. As we saw earlier, the production method that is capable of addressing the ruptures caused by expansion is thinking about teams in terms of their topology. The related terms, a scale-free or fractal system, coined by the management theorist Bill McKelvey to describe the system's self-preservation mechanism,[282] can be used to draw a parallel between the processes within complex systems understood more generally and the software production system behaviour in the context of continuous and unpredictable change introduced to it.

Following up on the fractality notion of post-ANT critique established in Chapter 2, the scale-free principle works with complex phenomena, such as systems, in a similar way through the patterns of self-equivalence across their various scales. Fractality, as Pierpaolo Andriani and Bill McKelvey explain, is the self-similarity of constituent parts that could be codified via the patterns that echo the whole, thus making it possible to trace their attributions.[283] Even though the agents have different tac-

---

[281] Marx, 1990: 724.

[282] McKelvey in Allen et al., 2011: 124.

[283] Andriani and McKelvey in Allen et al., 2011: 259.

tics in response to the stimuli unique to their contexts, what appears important to the scale-free construction is the matching strategic principle.

While Fig. 17 illustrates the assimilation of knowledge by the individual agent, Fig. 18 aims to clarify that a general principle of how the organisations and their parts relate to the software system remains unchanged in different levels of the production system. On the left of the diagram is the team level, where all of the agents are co-located in relation to one another and to the software system through the mediation of the problem space. Zooming out to the organisation level, the same pattern is present in the arrangement and interrelation between the teams. On a larger cross-organisational scale, the pattern is still unchanged, and all the organisations that use the same software system are linked in the same way. Arguably, the problem space universally connects not only the agents, teams and organisations but also all of these entities across the boundaries of organisations, as Chapter 4 deliberates in the discussion of distributed collectivities.

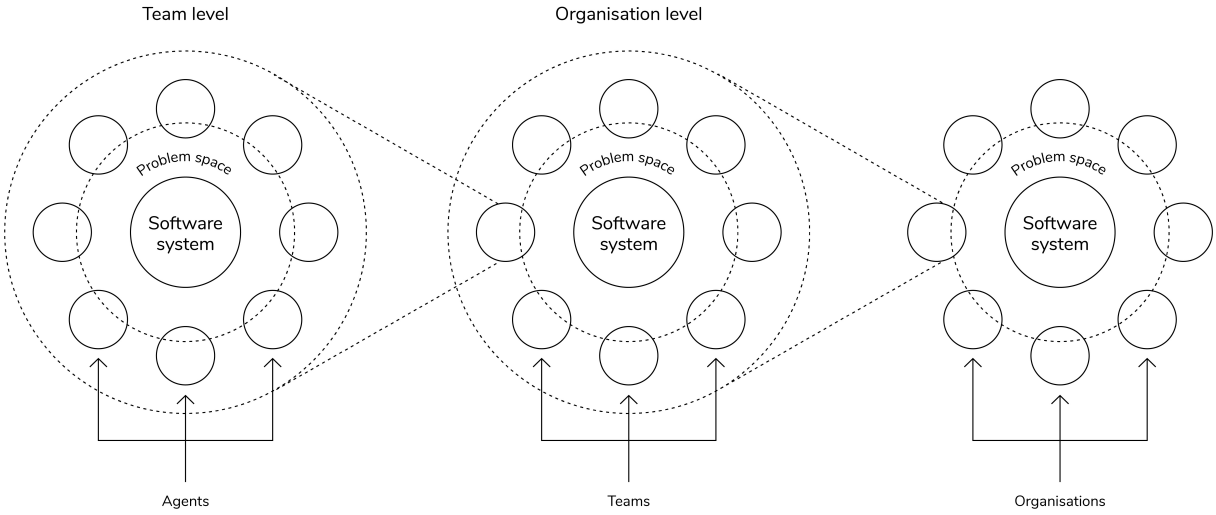

Fig. 18. The fractality of relations across the different organisational levels.

Translated into the terms of the stream-alignment paradigm, the general strategy described by the common organisation pattern is the key stream that the production team aligns to. Depending on the fluctuation of complexity, the team construction changes on a sliding scale, not dissimilar to the anti-fragile architecture of real-world buildings in the areas of seismic activity designed to absorb the impact of earthquakes. In the moments where the stream-aligned team encounters issues which it cannot address in its default shape, an incident is opened, and it gets rapidly associated with relevant non-aligned teams, and correspondingly returned to its usual shape after the incident has been resolved. All three types of non-aligned teams can be involved in different capacities – platform, subsystem and enabling teams.

The platform team works alongside the stream-aligned team and provides the supporting environment in which the stream runs, rather than the stream itself. It thus joins forces with stream-aligned teams in cases where the platform updates or other changes may interfere with the performance of the application layer. Enabling teams provide research services to guide the aligned team through any middle-scale complexity incidents. The subsystem team gets involved when the stream-aligned team encounters something that either was not a part of the system before, or something disruptively new that has never existed altogether. In this case, the complexity presents a real threat to system performance and thus the whole section of work is split out to be handled by a subsystem team that is focused on that particular technology, leaving the stream-aligned team to continue providing the service to the stream without interruption. Once the work on the complex component is done, it is either integrated into the stream with some additional retraining or reconfiguration of the stream-aligned team, or via the API, in which case the stream-aligned team never requires the new skills. In both cases, the complexity effects are exhausted before they have a chance to cause any stress in production. The scale-free approach circumvents complexity by limiting the agent involvement to simplifying the level of the abstract schema, regardless of how much change the system has to go through. What is required of the organisation is to develop a unified strategy and apply it throughout all levels, only focusing the maintenance efforts around the cases where the change dynamics cause inconsistent deviations. Just as long as the principle is homogeneously applied throughout the system, it seems enough for the auditor to use it to be able to review as many system parts as it is necessary, without an exponential increase in time and effort.

The parsimony principle of agent knowledge acquisition

The important factor in keeping things simple within the production environment is a trend towards parsimony that the agents undertake. Such a trend is related to the overarching reasoning informed by Ashby's requisite variety law discussed above and comes as a protective mechanism which aims to mitigate the risks of excessive data processing in cases when the problems present exponential computational demands.[284] It implies that throughout their dealings with the outside environment, the agents would always tend to find response tactics that would match as closely as possible the external impact. This implies that those systems that overreact and expend too many resources on inadequate responses head towards disintegration. Likewise, those systems that do not respond enough tend to gradually fossilise.[285] Agents achieve balance through the two-fold tactics of differentiating and integrating knowledge. The logic behind differentiation is that agents treat knowledge as a low-energy resource, which

---

[284] Boisot in Boisot et al., 2007: 156 citing Chaitin,Sipser.

[285] Boisot and MacMillan in Boisot et al., 2007: 65.

means that they tend to prioritise accumulating and using data, rather than more scarce resources of time, space and energy. Unlike the latter three, knowledge, once acquired, can be re-used without any losses to its quality or additional investment as many times as required, for as long as the inventory it pertains to continues to be relevant.

The use of knowledge, albeit less costly than more scarce resources, can be further optimised by differentiation, which the agents achieve by codifying it. Codification is a process of creating categories that classify the states of the world to better grasp the relationships between them. It aims to further drive down the computation costs by identifying the patterns within the incoming data that would allow splitting the information from noise. As Boisot observes, it works 'by extracting relevant information from data—that is, by exploiting and retaining whatever regularities are perceived to be present in the latter that would help to distinguish relevant phenomena from each other'[286] In other words, complexity is reduced by creating structures, locating the patterns that make it possible to create any degree of predictability. A well-codified knowledge inventory greatly reduces time spent looking for the required materials, however, can require a considerable upfront time investment. Staple production housekeeping practice such as writing technical documentation is a good example of codification that has to be carefully weighed against its potential future use. Quite often, the effort of writing up a comprehensive documentation can exceed its benefit, since much of what it describes can be learnt by using the functionality. Furthermore, if the feature is highly perishable, documenting it can be a waste of time altogether. In my fieldwork, I've come across a case where the documentation had been written for the library which later had been decommissioned in favour of a different one that had proven to be a better fit for the system. The lesson learnt, however, was that upfront spending is nevertheless worth it in most cases, for at least two reasons. On the one hand, it's best to document early while the memory is fresh. On the other hand, it is not entirely possible to predict if documentation might not be used, in any case, and thus can be a lesser risk than keeping knowledge in the undifferentiated form which might present a difficulty to make sense of later on.

The second parsimony move undertaken by the agents, integration, is based on *abstraction*, a term which is used here in the sense of treating things that are different as if they were the same.

Thinking about abstraction as the treatment of difference, such as in the view offered by philosopher and sociologist Alberto Toscano, may help to relate abstraction as a parsimonious move to Barad's

---

[286] Boisot and Li in Boisot et al., 2007: 88.

agential cut together-apart, which also thinks about the effects of difference.[287] To Toscano, abstraction helps to treat the difference between the multiple views of the phenomenon of high internal complexity by bringing together the effects of difference in the form of a unified determination.[288] Further adding Boisot's optic, abstraction can be seen as acting upon the codified results to create a tighter set of categories that pertain to a specific classification task at hand, and, as is the case with an abstraction of any other kind, it can be of a higher or lower level of generality, depending on the classification aims.[289] In the integration move, the previously differentiated knowledge inventory is mapped to the existing epistemic infrastructure. Where the knowledge and the infrastructure are incompatible, the infrastructure operates as a topological machine: it is viscous enough to be adjusted, but also able to condition the incoming data on its deeper abstraction layers – this is best illustrated with an infrastructure tool, such as Jira.

Atlassian Jira is versatile enough to provide a framework for opening support tickets and to set up the many fields and workflow stages each ticket will be differentiated by, and this plays a crucial role in both differentiation and integration processes. The main ticket types in Jira are stories and epics: epics are used to group stories together into larger shipments. Each of those types will use different screens to capture data. Similarly, workflows define which transition steps are appropriate for the story in consideration. Most workflows will have basic stages like To Do, In Progress, Blocked and Done, which, in turn, affects which screen and field configuration schemes are active. In the integration stage, the lower abstraction details of Jira can be adjusted – for example, if some additional fields are required to log in the project-specific information, they can be added via the Jira administrator interface. Some of the more broad infrastructural considerations, for example, the fact that the task has an 'Open' status when it is created and has to transition to 'In Progress' status to end in the 'Closed' status, is hardly a matter for debate or adjustment.

There are three interested parties in ticket writing, usually appointed in the industrial research as the firm, its employees, and its customers, and specified for the present study as production teams, business owners and users.[290] The Jira ticket is a contract that binds three types of parties together through the acts of gathering requirements, setting up the acceptance criteria and definition of customer value. Tickets as discrete entities also allow being grouped in different combinations, which may reveal various patterns within the problem space, which informs staff on planning, as well as opening very specif-

---

[287] Barad, 2010: 265.

[288] Toscano, 2008: 276.

[289] Boisot and Li in Boisot et al., 2007: 118.

[290] Callon in Law and Mol, 2002: 206.

ic avenues for audit. To retrieve the relevant knowledge, Jira provides a search field which goes beyond the usual graphic user interface (GUI) option and can be accessed through writing in Jira query language (JQL), which uses regular expressions and boolean logic similar to MySQL and other popular query languages.

The possibility of querying the problem space of production makes Jira a diffractive topological machine, in the sense that it provides conditions for automatic production of spatial orientation for problems described in JQL. To retrieve the topology of the problem space, a collection of support tickets based on any field value, parameter, condition or any combination of field values, parameters or conditions is stitched together as a continuous plane – or to use Barad's terms, the symbolic-material becoming of the problem is diffracted by cutting together-apart as a unified move. To begin with, the tickets are separated from the infrastructure that catalogues them, and then stitched back together according to the rules of their engagement and appear as search results. For example, the following query will return a list of bug-related tickets that Jira has records of, where the work either has or has not been started, arranged by their creation date:

```
issuetype in (Bug) and status in (Open, In Progress) order by Created DESC
```

The filtering, however, is only as good as its user experience and the attributes of the filtered content itself. Research using Jira has proven to be easier over the years of my fieldwork than any of its analogues, due to the accessibility of all the attributes, such as labels, statuses, component and cross-reference links, dates of creation and completion. All of them can be addressed in JQL, and thus, it is only a matter of Jira users' ability to log the issues with relevant input parameters that ensure the accessibility of the archives. With all of these parameters to consider, rigorous coordination between Jira and the negotiation channels of the problem space of production is paramount. For example, it is good practice to agree on Initiatives, Epics, Components and Labels on an organisation-wide level, so that JQL queries would return consistent results throughout the different organisation's departments. In other popular systems such as Notion or Trello the filtering is rudimentary to the point of frustration – for example, it is only possible to search the recently created tickets or by ticket name through the graphic user interface (GUI) with no option for regular expressions, which makes them much less accessible and therefore riskier to use in the situation of complex negotiations around the architectural and the political categories of the problem space composition.

Complexity and distributed governance

In correspondence to the previous section's assumption that organisations cannot evolve as quickly as the technology they utilise, this section aims to understand why, and what are the implications for gov-

ernance. To do that, the section starts by revisiting the notions of systemic change and momentum as the two counteracting organisational processes. Change and momentum, the two dynamics at the core of the production system in software capitalism, are taken here together as the two opposing forces that adaptive governance has to keep in balance to be able to maintain the system's traction. While change is something that enables the organisation to adapt to its competitive environment by introducing new parts to the production process, momentum is something that keeps the cohesion of its existing constituent parts. Upon further examination, it becomes clear that beyond the vested interests of stakeholders involved in the production, there are specific material conditions that resist rapid change.

Adding on to the discussion of momentum in Chapter 1, the two factors help elucidate its strategic meaning within the organisation. On the one hand, any decision that an organisation takes also narrows down its future strategic choices. Decisions are seen as long-lived assets, designed to bring value over a long period of time. After having taken a decision, the organisation tends to commit to it, further implicating it into its cultural and governance fabric. On the other hand, decisions have consequences for the regime of governance. Understanding those consequences may help clarify why the adoption of distributed management or self-organisation-based control models appears to be appropriate for creating auditable organisational environments and does not act instead to dismantle the production system altogether.

Momentum as resistance to change

In terms of decision-making, momentum is manifest as the tendency of commitments that the organisation takes upon itself to become a part of the protocol, which comes to shape its future strategy. The change dynamic is not necessarily the problem in itself; however, as Max Boisot observes, it contains a risk because the change is often path-dependent and irreversible.[291] What this means is that every decision that the system takes acts to close off the possibilities for some of the alternative scenarios, as they become too costly or come into conflict with the effects of the previous change. Herbert Simon describes such commitments as *sunk costs*, in view of which any rapid adjustments are not as profitable as staying with one decision, which usually promises long-term gains.[292] The fallacy of sunk costs, to Simon, makes the investments made in the past a background that the new decisions have to be evaluated against,[293] or, in other words, under the increasing weight of past decisions, the system proceeds along a certain path that makes it further determined in a specific direction. Furthermore, often

---

[291] Boisot and Li in Boisot et al., 2007: 77.

[292] Simon, 1997: 148.

[293] Simon, 1997: 148.

changes cannot be reversed, which may cause a problem in production when the changes cause system failure, or on the side of management, who anticipate the failure and are reluctant to make the change, preferring to postpone the decision instead. Putting the decision off to a later moment usually assumes that the decision can lead to good outcomes under favourable conditions, and to poor outcomes when the conditions are not that supportive.[294] In the context of rapid and radical change, however, no easy assumption can be made as to whether the environment will grow more hostile or more favourable in the future, and therefore is no lesser risk, particularly if there is no wider environment forecasting programme.

In the second place, there are real-life factors that make formal architecture more resistant to change as compared to strategy. This kind of resistance accounts for the time and effort it takes to actually implement the change – for example, any practical human resource dealings with staff hired on various contracts, but also the time it takes for the staff's personal beliefs, including shared organisational values, to accommodate the policy updates. Even after any changes to the strategy are written down, the cultural trends take time to adopt them. Simon comments on this that during its operation, each organisation acquires *sunk assets*, which can come in the form of specific know-how – 'the way we do things around here', which comes together with goodwill – community-like relations between staff that aids communication, and is not easily transferable into a different activity.[295] The combination of these factors creates a particular implied notion of momentum, which might not be made explicit in the organisation's policies or technical documentation, but instead tacitly accumulates over time to create a shared understanding that any change in objectives would entail a decrease in efficiency associated with loss in some of the sunk costs or assets, as well as a potential erosion of goodwill. Such understanding of an organisation's inertia gives a new meaning to the interpretation discussed in Chapter 1 because, rather than thinking about how technology shapes society, or in this case, the organisation's production lifecycle, it aims to understand in which way the inability to communicate the knowledge is treated by the organisation, and what kind of risk it presents. Such knowledge is often referred to as tacit, and a more detailed description of this notion at this point will help clarify the character of the organisation's treatment of it.

Momentum and tacit knowledge

Within the production lifecycle, information is located explicitly in the form of documentation, which is meant to reflect the epistemic infrastructure but may have gaps depending on how much priority the

---

[294] Ibid.: 137.

[295] Ibid.: 148.

organisation gives to documentation work. Information implicitly embedded in the components of the infrastructure itself can be more reliable. Agents apply conceptual filters, such as specific context informed by the support ticket, when using the information found in the organisation of the system's environments, in its configurations, in the source code comments or in anything contained in the code repository. In their engagements with the epistemic infrastructure, the agents accumulate or adjust the expectations they may have about the kinds of knowledge they gain about the system. Lastly, as the agents form a specific relationship with the kinds of data and information they usually encounter in the system, they acquire a mental model that enables them to create the situated knowledge, based on the expectations.

Faced with a change that comes from outside of the organisation's sphere of control causes a loss in momentum because the culturally constituted ways of doing things with the specific combination of the organisation's skills and technology are no longer seen as viable in the new production context. As we saw previously, momentum is instructed, on the one hand by the business requirement of getting the software product out into the market as soon as possible, and on the other hand by the efforts of corporate governance to pull the controlling protocols together so that it continues to be possible to manage the risks and ensure that no regulations are violated or that no quality issues lead to the loss in sales. The key issue is with maintaining the balance between the parts, whichever condition they are in. An important factor to momentum is that it is something that the whole organisation works hard to build up, often, as the case study CS15 shows, as part of establishing the basic initial communication patterns. Thomas P. Hughes emphasises that at any given point in the project, the degree of its success is informed by the degree that the various parties are invested in its different aspects. This means that any organisational initiative is underpinned by the web of interests, including funding bureaucracies, engineering leads, and any executive or trustee boards that are going to be affected by the changes in production. Therefore, it is only natural that where a certain existing momentum already warrants predictable outcomes, the disruption in the way of things can be met with resistance. While the momentum may offer rapid execution along the well-trodden paths, the path dependency is precisely what presents a problem when dominant logic is confronted with ongoing change. As Max Boisot demonstrates, such logic introduces friction that makes the cognitive switches harder to achieve, and while the persistence of inertia may look like a benefit from a neoclassical economic standpoint, in the face of emergent qualities of the problem space it is, in fact, a non-equilibrium phenomenon.[296]

---

[296] Boisot and Li in Boisot et al., 2007: 97.

The innovation proposals that the existing institutional framework is capable of resisting are something that can enter the organisational imagination in some form – there could be cost estimates, roadmaps or architectural blueprints. The extreme organisational rupture is caused by more serious tectonic shifts that are not proposed or charted at the inter-organisational level, simply because the construction of the problem space does not account for something that lies outside of the problems it had been used to negotiate up to the present moment. There are no prior givens, goals and operators that can be utilised to tackle radically new problems, and therefore the activities of problem space of production become entangled with the abductive leaps that are taken as the last resort of bridging the gaps of insufficient knowledge. However, as this chapter's earlier discussion of cognitive load demonstrates, depending on how large the unknown terrain is, abductive leaps may take considerable time to investigate the new contours of problem space. The issue is, that in situations of extreme complexity, the scales of innovation can be so vast that the search time takes long enough for the system to lose traction where it previously had it. In this case, extreme organisational rupture occurs, leading to the inability to move forward without a substantial re-organisation.

While the situated knowledge factor accounts for the topological quality of knowledge, its tacit quality points to the degree of its entanglement in production practices. To the philosopher and economist Michael Polanyi, who closely engaged with the notion, tacit knowledge always involves more than anyone can say and is to a greater or lesser degree enmeshed in skills and know-how. It is central to the problem to have an element of discovery, 'the intimation of something hidden.'[297] In other words, tacit knowledge benefits the affective side of communication between individuals involved in production in the same way as explicit knowledge benefits effective business operation. Since tacit knowledge is personal, context-specific and derived from direct experience, it is inseparable from communication, which is often an integral part of practice rather than verbal or expressible by other external means. The association with practice makes the notion of tacit knowledge important for my argument since implied property means there is no associated effort of making it external for production, which, in turn, means that the use of tacit knowledge adds a great deal of momentum. The knowledge which is tacitly present as part of daily production practice does not require meetings, onboarding, emails or technical documentation. It is simultaneously a part of the organisation's design and its culture.

For this reason, tacit knowledge enables the parsimony tendency within the agent's behaviour, in the sense discussed in the previous section, as something that does not require integration efforts. In some instances, tacit knowledge does not have to be something that cannot be communicated – for example,

---

[297] Polanyi, 1983: 22.

where participants are placed within the same context, the knowledge that could otherwise be written down in the company wiki or sent via chat or email, under the circumstances can remain unsaid at all. The knowledge that remains unsaid, however, also presents a potential business risk since the inability to communicate, which is a benefit in the existing context, may appear as a bottleneck if the workflow is changed, or people are no longer involved with the business. This means that such momentum of Simon's sunk asset of *goodwill* should, as we saw in the description of Fuller and Goffey, be met with resistance by the organisation, as a hindrance to the circulation of knowledge assets and adds to the expenses of differentiation and codification.[298] This additional benefit of tacit knowledge to operations is not always recognised on the executive level, which may have a hiring strategy that does not account for it. The case study CS15 – an archival project case study in the Appendix – illustrates the extreme rupture of the problem space of production that was caused by another reason, the change in geopolitics, which, however, has had a similarly disruptive effect on the production process, and therefore can be looked at in this context.[299]

The case study looks at the situation that occurred in the aftermath of the abrupt halt of the production process of the publication (JX) in early 2022. The stop press had been made necessary by the changes in the company's strategy which were caused by the Russian invasion of Ukraine in that year and the rapid severance of international economic and cultural links across the region. Six months after ceasing publication, JX came up with the requirement to redesign the system for the new purpose and re-launch what used to be a media channel in the form of a digital archive. I was involved as the project lead both with some of the old team and new staff hired for the production of the archive project. One of the challenges, in this case, was that when a part of the old team was gone, a good deal of the production momentum was lost. Even in the presence of documentation and the continuing support from myself as the bearer of much of the previous production knowledge, the new team members had to begin by applying conventional thinking to the system which to them was completely new. This has led to some frustration and misunderstanding, initially from my side, since when writing the briefs and commissioning designs, the idea was to save the effort and to re-utilise the existing system components. The lack of momentum in the new team, however, meant that it was not possible to simply start building where the old production team left off. Instead, lots of seemingly ready-to-use components, such as the top navigation, appeared easier to recreate from scratch, due to the now lost situatedness of the knowledge about how this component was embedded in the overall structure of the system. Having to recreate the components from scratch led to increases in the initial phases of the

---

[298] Fuller and Goffey, 2012: 127.

[299] See Appendix CS15.

project, however as the system-specific mental model began to take shape in the minds of the new team members, the momentum had noticeably built up.

Empirical cases as above have instructed me on the field uses of abductive modelling as it develops in conjunction with creating the body of tacit knowledge. In situations of extreme organisational rupture, the agents have no other option but to go beyond the available evidence and start by generating hypotheses. To Magnani, whenever the stable ontological grounds of reasoning are shaken, the hypotheses tend to transcend the existing agreements between the paradigms, as in the scientific discoveries during the transition from classical to quantum mechanics, which were made through the use of abductive reasoning.[300] In the realm of production, the abductive mechanisms are used in teams as a rule of thumb without much theorising, but rather with the aim to reach the point at which the hypotheses are developed enough to start the testing iterations. To maintain traction, the abduction needs to take into account any existing components, such as databases or environments. Databases are usually robust enough to survive complexity spikes and are expected to be highly regular, subject to the efforts of the system's database architects, however, can be highly opinionated. In one of the field cases, I learnt that the out-of-the-box CMS, including the one used at JX, are usually not fit for most production environments not native to them, since they came with pre-designed databases that fit their original needs, and are usually incompatible for other users whose needs might differ. In other words, every production system comes with its own assumptions about the format in which the data comes in. While the complexities of such kind are accidental, they may allow making a case that the accidental complexity is where the most crucial complexity cases may often be found.

Concluding the section on change and resistance, it should be noted that while every change may meet resistance within the existing organisational structure, it is particularly important to account for the changes that the system is not prepared for, due to the inability to account for them, since such changes have more potential to bring the system into misbalance. An organisation may find itself under pressure to change in response to the changes that occur outside of the production context under consideration, and therefore without any coordination with the organisation's own governance or audit planning. For example, change may occur in the wider ecology of the production system, such as updates to the operating system or third-party or open-source libraries, or changes may occur in the market domain caused by the adoption of the new technology by the competitors.

---

[300] Magnani, 2009: 33.

Returning to the two examples we saw earlier in this chapter, the AWS migration project and the Archive project carried out by the new team after the project was officially closed, can be seen in this context in terms of their responses to the radical external changes. In the former case, because the migration was performed as part of the crisis mitigation measures, it was carried out without particular regard to the organisation's overall internal planning. This had placed the additional complexity stress on its production system because the utilisation of the CloudFormation, the infrastructure as code service which comes as part of the AWS offering, is generally associated with a more specialised DevOps treatment compared to the system JX used before. This, in turn, implied an additional cost consideration, which in this case could not have been done in advance, and therefore caused tensions in the organisation's production budget planning.[301] In the latter case, where the team cohesion was ruptured in relation to the major and long-lasting geopolitical trend, the primary focus has been on building the body of tacit knowledge based on technical documentation and gaining enough traction to deliver the required updates to the product while trying to abstract, circumvent or delay any further complexity dealings, such as code review, refactoring and regression testing to a later stage.[302]

Specifics of coordination in adaptive complex processes

As the discussion of the cognitive load suggested earlier, due to the limits of their capacity to assimilate new knowledge, the agents seek to position themselves out of harm's way of complexity. Beyond the tactics of differentiation and integration, the agents tend to further reduce the complexity effects by employing the distributed principle in their processing efforts. As Max Boisot explains, the distributed character of processing means that the agents, otherwise scattered within the boundaries of the same problem space or the same domain, come together to participate in the event of problem negotiation on a case-by-case basis. Furthermore, distributed cognition works equally well both in homogenous and heterogenous epistemic environments, meaning that it is not uniquely linked to the common knowledge that the agents may have, but also to the differences in their knowledge – the inconsistencies here facilitate self-organisation since the diversity of understanding of the issue at hand may lead to faster problem-solving.[303] The activity of abstraction that the agents carry out is differential, meaning that it is not intended to express any generic objects suspended from differences, and is close to Marx's real abstraction, which, as we saw in Toscano earlier, arises from the varied determinations of agents in the historically specific relation of production.[304] The agent coordination that is based on their differ-

---

[301] See Appendix, CS12.

[302] See Appendix, CS15.

[303] Boisot and Li in Boisot et al., 2007: 101 citing Hayek.

[304] Toscano, 2008: 275, 277.

ences makes it imperative that the agents discern the patterns within each other's behaviours to maintain the group cohesion, as well as the patterns of their environment. The regime of coordination that enacts the rules for such mutual adaptation should therefore be more precisely defined as adaptive governance. Such governance is characteristically non-centralised and scale-free.

The rules enacted by adaptive governance comprise something that can be defined as *protocol*, a set of soft regulatory principles that do not imply administrative compliance, but rather define a management style and bear cultural value as something which is shared across the organisation and on a wider scale of a community of practice. The protocol's relaxed applicability is necessary for it to be relevant in the uncertainty of complex production situations. More specifically, the advantage of the protocol is that it can be applied to both the practice and process. The practice is viewed by the administration as the enactment of situated knowledge accumulated in a specific community, where its members continue searching for new solutions within the boundaries of their domain. The process, in contrast, is the vertical spread of knowledge, which cuts through the various levels of situated knowledges. To illustrate the difference, it is worth evoking Adam Smith's rich example of a pin factory one more time, albeit for a different insight. As we saw previously, the protocol of the factory breaks down the production of a single pin into the activities of separate workers, such as drawing out the wire, straightening it, cutting it, and so forth. The benefit of the division to practice is that each individual worker, once relieved from having to switch between different activities, can creatively explore and optimise the specific activity assigned to them. In terms of process, the pin as a result of collective effort is present as the guiding principle for all the subordinate production events and as a tool for quality assurance. In complexity management scholarship, it is usually acknowledged that away from the traditional centrally controlled manufacturing operations, most systems today, including the ones involved in complex production scenarios, are managed in a distributed way as clusters of local practices, while remaining auditable as processes through the main guiding principles.

More specifically, distributed governance becomes possible when three factors pointed out by Maguire, Allen and McKelvey are present: readiness for organisations to enter into a coordinated relationship, their sufficient connectivity that makes the mutual coordination possible, and abundant resources.[305] The distributed system can thus be defined, via Azadegan and Dooley, as a system where there is no one source of ultimate authority, or even in the presence of a protocol it is the agent body that bears the most responsibility for decisions: 'in a distributed control system a number of agents are respons-

[305] Maguire, Allen and McKelvey in Allen et al., 2011: 15.

ible for sensing, interpreting, and controlling actions.'[306] In real-world production, for example, as I discovered in my fieldwork, the scenarios were mostly mixed, with main strategic decisions coming from the key stakeholder, which was then met with resistance from the self-organised agent groups who may have found the decisions incompatible with the accumulated local knowledges. As the production process continued, the outcomes were further evaluated against the strategy, and the strategy would be adjusted, resulting in negotiated solutions that would allow effective delivery for everyone involved.

Chapter conclusion

This closing chapter of the thesis has looked at the governance implications for complex responsive production systems. The governance here appeared as distributed and operating not through any fixed hierarchy, but via the audit applied throughout the product system in a scale-free way. Such audit is necessary, in the context of complexity's fluctuations, due to its ability to circumvent them by limiting its inspection to the organisation's internal managerial structures, which, in turn, report on the performance of the self-organised agents' associations. The chapter has explained such fluctuations as a necessary feature of software capitalism that comes from its fundamental tendency for the cost of computation to fall. Complex adaptive systems avoid internal hierarchy by maintaining instead a protocol, or a unified set of principles, for codification and abstraction. The protocol is cultural in that it springs from the beliefs shared by agents throughout the organisations and communities of practice about the ways in which the knowledge should be classified so that it could be searched, sorted and filtered by others later. The shared codification and abstraction beliefs, paradoxically, preclude the integration of knowledge due to the inevitable fact that some of the knowledge is present in its tacit form.

Therefore, the more complexity the distributed governance is presented with, the more non-transferrable tacit knowledge is generated by the agents locally, and, concomitantly, the more resistance to audit there is. While in this case, there might seem like a semblance between the centralised and distributed systems, it occurs for different reasons. In the centralised hierarchies, the inability to audit happens due to the presence of real-world limits on how much information the system can process and transmit. In the complex adaptive system, conversely, there is no problem with overflows and the stifling of reporting capacity here is intentional and, to use the Marxian term, appears as a feature of real subsumption of labour, that is, fully conformed and appropriated by the software capitalism production process. Tacit knowledge is produced as a means of creating scarcity within the system that otherwise wouldn't have it due to its fractal character. While in its tacit form, the knowledge is difficult to scale across the system in the event of a complexity spike, it provides a great return on investment in

---

[306] Azadegan and Dooley in Allen et al., 2011: 421 citing Tunalv, Radner, Deshmukh et al.

terms of efficiency increase warranted by the momentum. When the agent's abilities to codify and abstract are overwhelmed, they suffer cognitive shocks, causing losses in efficiency. At this moment, the knowledge is converted to its explicit form, making it possible for the organisation to create new business divisions and address any complexity effects. The capitalist valorisation mechanism, in turn, propagates to the team and organisation levels by increasing capital circulation in those new divisions. As the discussion throughout the chapters of this thesis as a whole has aimed to demonstrate, the new divisions may not have been required without the situation of artificial scarcity created for capital circulation.

General conclusion

In conclusion, the study looks back at the argument as a whole to discuss in which ways the theory that I engage with can be seen differently in terms of the present discussion. The sources this research deals with can be generally split into three types: background, data and focal theory. In terms of the background theory, the research generally assumes, in its interpretation of the socio-political conjunction, that the software capitalist production system finds itself in, the fundamental categories of the labour theory of value theorised by Karl Marx. The other resource is the systems theory thinking of Thomas P. Hughes, which provides the context for understanding some of the characteristics of technological systems, such as momentum and the specificity of the system's audit. Next, Melvin Conway's organisation design paper plays an important role in current DevOps thinking and in this sense is an important background source that creates a link to the industry, which the next category of sources looks at.

The data sources in this thesis are the DevOps professional literature and other associated references that I use for deriving the industry data for this study. The source that underpinned the initial phase of my research is the writing of Frederick Brooks, in which I find historical evidence on the pre-Agile era strategies for mitigating the software crisis. Since a lot has changed from the time of Brooks's original writing, I use some of his claims, such as *invisibility* or *conceptual integrity,* as the questions to approach the more recent industry research. In terms of present-day sources, I clarify the Continuous Delivery and deployment pipeline using the work of David Farley, the author of these production concepts. Next is Matthew Skelton's proposition of a stream-aligned production pattern, which applies some of Conway's findings in the practical operations work. During the research, my view has shifted to understanding the term *team topology* as a more general analytical model, while referring to Skelton's *team topology* more specifically as the stream-aligned paradigm. Lastly, Gene Kim's nuanced account of current DevOps standards helped to articulate the concept of the coincidence of business value and an organisation's technology stream. Beyond the description of practice found in theory, I'm also motivated to include the case studies I carried out in the field as additional sources of data theory. The empirical engagements have established the links between the abstract definitions of practice and the concrete practices as they take place within the organisation.

The focal theory of this study reflects its main interest in infrastructural qualities of epistemology, with the aim of understanding how knowledge is mobilised in a productive assemblage for control and

planning in situations of exponential complexity increases. Such a goal steers the overall strategic direction of this thesis towards the inquiry into method, and, symptomatically, the focal theory sources are predominantly methodological. The key reference here is Celia Lury's compositional methodology framework – the epistemic infrastructure and the problem space – are used for thinking about software production systems. They are compatible due to their focus on the negotiation of problems in an incremental way, which applies to the production of complex systems, and has, in fact, been used in software production since early Agile. The other focal source is the agential realism philosophy of Karen Barad, which relates to the production of software systems as a process of negotiation of meanings and extends the understanding of such negotiation through a diffractive view of the problem space of production that implicates it with materiality, agency, discursive practices and causality. Despite the strong ontological orientation of Barad's work, such as in her theorisation of the agential realism framework, a relational ontology which focuses on the mutual becoming of material-discursive formations, I find that methodologically the framework of Barad is compatible with my epistemological inquiry. It specifically instructs me on the possibility of applying diffraction in software studies as a way of queering the DevOps treatment of the knowable and the knowing matters by looking at it through the optics of embodied practice and affect. Lastly, I engage with the abductive modelling method proposed by Lorenzo Magnani to create, quite literally, the production lifecycle model as a blueprint for my research, employing abduction as an operative principle of iterative composition of the system alongside the knowledge about the system. The model, however, necessarily borrows from the fourth focal theory source, Max Boisot's assimilation of knowledge as strategic cognition, which is required to understand the limits of traction and how momentum is made possible on the level of agents.

Focusing specifically on the two groups of sources, background and focal theory, this Conclusion is split into two corresponding sections. Each of these sections looks back to the theory the thesis engages with and summarises the insights that could be derived from the discussions contained throughout the thesis chapters. The purpose of such a cursory overview is not so much to describe the work that has been done, but rather to activate the content, terminology and lessons learnt for future research.

Background theories

Scalability of means of production and distribution

Turning to the background theories of the present study, several considerations have had to be taken into account when thinking of the distribution and production mechanisms through the optics of the labour theory of value and the management studies informed by complexity thinking. The most prominent characteristic here is that the capitalist model tends to valorise the complexity through scaling, which is made possible by the benefit of the reuse of virtual inventory. Despite its tendency to rapidly

become obsolete, the inventory in the context of software capitalism, does not perish in the same sense as in material logistics, through reducing or losing any of its qualities in transit or repeated use. Moreover, the distribution of virtual inventory may come at a considerably lower cost compared to the distribution via the material channels. The inventory itself is not limited to the outcomes of production, but is also present in the form of databases, environments or configurations, which are more often used as means of production or distribution, yet are also re-used indefinitely without perishing or expiring because of how often or intensively they are utilised. To business owners, in other words, there is a scalability benefit, which does not only apply to the outcomes of production but extends to the means of production and distribution. This implies a wider potential for scaling through inventory reuse, which is only limited by the compatibility between the systems, their components and the hardware they run on. For example, the material infrastructures of the internet are susceptible to deterioration, and in that sense can be compared, by way of Nadia Eghbal's allegory, to real-world roads and bridges.[307] On the contrary, code-based technologies such as Apache or NGINX web servers, which carry out the work comparable to substations of the electricity networks, do not suffer wear and tear in quite the same way as their material counterparts.

Requisite variety effects in organisation design and the practices of audit

Turning to the issues of production system scalability, I find that these involve production practices as well as the practices of audit, due to the tendency of the capitalist mode of production to reproduce and enforce its constitutive social conditions. The two forces amplify and promote this tendency in the context of software capitalism: the cybernetic principle of requisite variety and the depreciating computation costs. The former makes it imperative to respond to the environment in an equally complex way, while the latter makes it possible to scale rapidly while doing so. As a result, the market tends to intensify the capital circulation by instructing the software system to become more complex and the organisation responds by adapting its structure to replicate these components. Such replication, however, becomes increasingly difficult in systems of high complexity, since the software system is capable of scaling faster than the organisation and can present its agents with overwhelming amounts of stimuli across many of its abstraction layers simultaneously.

For production teams, this means that cooperation no longer brings free gifts to capital in terms of added effectiveness or faster production. The effectiveness does not come for granted because adding staff members, in agreement with Brooks's Law, provokes a *tar pit phenomenon* – an exponential increase in the efforts of communication and learning. Since these are not scalable, they create, at best, un-

---

[307] Eghbal, 2016: 8.

resolvable bottlenecks in production throughput. The situation is aggravated by the various industry antipatterns which assume that the tar pit phenomenon can be overcome through high utilisation, large batches, forward planning or adding new features. As Chapter 4 illustrated, however, the communication bottlenecks largely concern the centrally-controlled production configurations and can be avoided by more flexible organisation designs, such as stream-aligned team topology. Concomitantly, there is an equally important paradigm shift in the audit, which is understood as the formal practice of self-observation that, unlike the production events, are a part of the production system itself and are therefore reproduced as part of the conditions of production. This presents the auditors with a unique set of challenges, discussed in Chapter 4, to reconsider audit as a distributed practice.

Valorisation of complexity during the integration of knowledge

The third aspect deals with the capitalist valorisation of complexity that emerges around the event of integration of knowledge. The valorisation here is connected to the additional possibility of qualitative systemic change that the software system is open to since each instance of change causes complexity which has to be mitigated. As the chapters throughout the present study aimed to demonstrate, there is more potential for complexity in the production of digital knowledge-based artefacts than there is in the manufacture of material goods. The reason is that, however complex the material artefacts may be, they will always be constituted by a finite number of parts, which in turn puts a limit on how much quantitative or qualitative change can be made at the factory in any time period. In contrast, in software production systems, the radical changes with ensuing spikes of complexity are a risk which is constantly present. Quantitatively, the system may experience an increase in the incoming user traffic, for any external reason that has nothing to do with the system itself, which risks overwhelming its capacity for performing repeated operations. Depending on the system's architecture, the quantitative spike could result in either system response timeouts or an increase in compute bills from the provider who has to serve more requests. Qualitatively, the system risks facing changes in any of its abstraction layers, where some of these changes may occur outside of the organisation's control. For example, the fact that most software products rely on external libraries or plugins may present a risk, since any components located outside of the organisation can introduce changes without being able to consider the conflict in all the downstream components and services that use them. Such qualitative change, due to the likelihood of complexity arising from the inchoate, uncertain and unprecedented events that are not yet comprehended enough to be articulated, tends to inhabit the sphere of the organisation's tacit knowledge. Quantitative knowledge tends more towards an explicit kind.

Each integration event triggered by change is a knowledge transaction that presents an opportunity to extract circulation profits. Because of the above, this can be done in two general ways. On the one

hand, it is by preventing the knowledge from being made explicit, and on the other hand, once it does become explicit, using it to scale the production to expand the circulation of knowledge. The reason why the latter does not happen in all cases is that scalability is only possible based on the knowledge that can be circulated universally through the infrastructure, code or technical documentation, and not the knowledge which is only present in its tacit form, such as embedded in the human practices. Tacit knowledge is a product of the system's momentum and therefore always abundantly exists within the system, bearing the benefit of increasing the production speed in the local context when the agents have been working together for a long time and therefore have a lot of shared knowledge which has no reason to be integrated into the explicit form. Simultaneously, this means that tacit knowledge is a scarce resource since it cannot be effortlessly communicated to new teams in the event of scaling. The valorisation of tacit knowledge in this sense implies that the levels of tacit knowledge are always kept up to the level of agents' breakdown from cognitive shocks, at which point it gets integrated into explicit knowledge which is higher in scalability, albeit slower in production.

Focal theories

Presenting the categories of compositional methodology in DevOps

The present thesis proposes to present the two key categories of compositional methodology, the epistemic infrastructure and problem space in relation to the Continuous Deployment paradigm, to be categorically compatible with the audit carried out in the production context informed by DevOps. The former term, therefore, appears as the *epistemic infrastructure as code*, and the latter as *the problem space of production*. In their adjusted definitions, the terms similarly assume the respective roles of the two domains in the process of negotiation of meanings within the production lifecycle. What makes them more suited to the study of operations is that they become closer to dealing with the software capitalism formation, characterised by the equivalence of the business value stream to the technology value stream. On the one hand, the identification of the problem space as the necessary attribute of the production process positions it in the boundary between the market, where the outcomes of the software capitalist mode of production are realised, and the problem space of organisational culture. Another aspect of the problem space *qua* production space that the category reflects is its function as the meeting place of the organisation and the community of practice in terms of their relation to the software system. As Chapters 4 and 5 discuss, the problem space of production offers the opportunity to negotiate the meanings of the system criteria, and it does so in a scale-free way due to its topological presence in all of the system's abstraction layers. While the present study has examined predominantly the stakeholder types of the agent layer, such as business owners, production staff and users, the negotiations similarly occur on other scales between the teams and whole organisations.

On the other hand, *epistemic infrastructure as code* replicates the infrastructure of knowledge in the code in the same way as DevOps replicates the infrastructure of the software production system in its version control. This permits the epistemology of production to communicate with the process of knowledge integration as a necessary reproductive dynamic of the business value stream and audit. The study of knowledge infrastructure as code pursues similar aims to the creation of the deployable infrastructure in the Continuous Deployment method in DevOps. Both phenomena allow tracking changes and to be able to track when the changes were made and by which individual team members. Infrastructure, through being traceable as any other application code, provides a set of rules that warrant the control of other controlling mechanisms. In this sense, studying the infrastructure as code does not merely contribute to the study of audit practices that pertain to that code – it is, in fact, the study of such practices.

Conflicts in abductive models on different levels of abstraction

The abductive modelling interpretation of compositional method (CM) in the present study meant that any models would have to deal with the problematics of the epistemic infrastructure and the problem space, as the two key categories of CM. The model itself became necessary given my hypothesis that the knowledge that informs the production context, due to its parsimonious treatment, is continuously undergoing the processes of codification and abstraction. Therefore, it must be using some sort of epistemic mediator to keep the circulation coherent and within the bounds of a system. In the next step, I have outlined a diagram for the epistemic mediation, featured in Fig. 4 in Chapter 2. This has revealed the production system as an interface which creates, by internally circulating knowledge, a possibility for material, performative and affective negotiations between the market and the organisation domains. During fieldwork, however, the theoretical moves of deployment and integration got entangled in the empirical organisational context, which suggested that the abductive reasoning of more than one vantage point has to be considered, which led to the creation of additional diagrams.

The production pipeline diagram, as per Fig. 2 in Chapter 1, deals with the activities of the individual agents within the production team, making it possible to open up their cognitive processes for symbolic analysis. Taken together with the integration process diagram in Fig. 17, and the data collection flow in Fig. 21, it sheds light on the localisation tendency, which means that agents tend to avoid the overwhelmingly complex production with its many aspects by prioritising the communications to their immediate neighbours. This tactic provides some protection from cognitive shocks by limiting the information to what is shared between the agents located close to one another. The local assimilation and interpretation of knowledge, in turn, may create differences in meanings, since the agents negotiate to take phenomena they encounter for what they are in specific production contexts, rather than through

the lens of any centrally instructed strategy. As Chapter 5 observes, this locality is usually seen as the benefit of the adaptive construction of a system, which empowers the agents to specialise and diversify their behaviours to find creative solutions.

Yet, once the scale is shifted as per Fig. 18 which shows the self-similarity of the relation schema on the various scales, the diversification on the lower levels may mean conflicts in the higher levels – for example, incompatibility of the locally constructed meanings with the audit protocol. The conflict may suggest that abductive manipulation of production models happens differently at different levels of abstraction. As I have encountered in the field, the local decisions were prioritised due to the complexity of production situations, but there was always a limit to how distributed the system was allowed to grow before it started to interfere with the coherence of the organisation's strategy. To echo the observation that Chapter 5 makes in this regard, where the abstraction level is high enough for the complexity to be handled by the centralised form of governance, such a form would always take precedence over self-organisation, and the local decisions would no longer be prioritised.

Queering the DevOps epistemology

While the reason for choosing software as the matter of present study is that it provides the appropriately complex production context, the reason for prioritising the DevOps optic, over, for example, that of programming languages, is that it provides unique access to the epistemology of the software production system operations. Epistemology in DevOps, as something that contributes to the method by informing the research not on the content of specific cases, but on how the content is created and organised, is a central concern to operations research as a place where the system integrates the effects of its complexity. Furthermore, operations deal in equal measure with the market, and the organisational aspects of the system. The multiple frictions between the two domains are being negotiated in production, which appears for this purpose as the interface. This means that many production practices, of which the present thesis takes an example of Continuous Deployment discussed in Chapter 3, are, in fact, responsible for shaping the boundary between the market and the organisation, and being shaped by it in return. For example, Continuous Deployment is simultaneously production, in that it creates the value-bearing commodities ready for exchange, and yet it is also distribution, in that it makes the outcomes of production available to the users.

In this situation of uncertainty, the use of diffraction when looking at production makes it possible to simultaneously arrive at the same problem from multiple viewpoints. This would then allow engaging in problem composition based on the effects of difference between these views. The support ticket, as Chapter 4 aims to demonstrate, occupies a central methodological position in the queer view – that is,

a view which is capable of interpreting the system in terms of its dysfunctionality, and of tethering the stakeholders to the relevant criteria of the problem space of production for the duration of the problem-specific application of method. Support tickets make the multiple readings of the problem available for queer DevOps manipulation, which is simultaneously symbolic and material because it deals with representations and yet bears real effects in the world and lived experiences.

Some of the other important boundary categories that appear simultaneously in different capacities are the categories of production participants, referred to in my research as the empirical and the epistemic. As Chapter 5 notes, the participants are split into the owners of the means of production and the wage-labourers, the categories that can be used to study their involvement in the relations of market value exchanges. Yet, the groups are also simultaneously unified as the stakeholders of the agent-based presentation of the production system, which pursues the goal of establishing a web of inter-organisational relations that operate through the functions of administration and therefore pursue compliance, rather than the surplus value per se. The consequences of such simultaneous enactment of different capacities are similar to the ones Anna Tsing arrives at in her concept of supply chain capitalism. Where the latter interprets cultural dimensions such as gender, race, ethnicity, religious identification, age or citizenship as the niche banished from the economic, which is nevertheless used to vitalise the class relation for mobilisation of labour,[308] the diffractive view of class and stakeholder relations similarly conceals the alienation and exploitation processes within the specific aspects of organisational culture. While the market activates specific mechanisms that allow it to appropriate the cultural dimensions of production, I argue, the organisational teleology acts as its subset and organises its audit and other control mechanisms in alignment with the market consideration, albeit often in a non-obvious way. For a more nuanced discussion of the interrelations between the epistemic and empirical categories of production participants, it is necessary to conduct a diffractive analysis of the relation of class, which lies outside of the scope of the present study.

Another category that comes under the diffractive lens to consider the differences it deals with is audit. What could be learnt from Chapter 4 is that in complex systems, audit can only be present in its local form, being situated in a specific organisational context, and can only operate on the organisation's internal managerial structure. Locality here means a way of conducting audit without the attempt to control its every aspect from the administrative centre. The distribution of efforts throughout the multiplicity of localities allows positioning the control mechanisms orthogonally to the force of unbounded system complexity, to be able to withstand its potentially infinite fluctuations in a scale-free way. The

---

[308] Tsing, 2009: 158.

distributed form of audit escapes complexity by taking advantage of abstraction layering, which in software capitalism depends in equal measure on the two kinds of abstraction. On the one hand, there is software abstraction in its technical meaning used in computer science: the hiding of code. On the other hand, there is Marx's real abstraction which Alberto Toscano has arising from the real world of the disparate agent determinations, which are interpreted in a strategic formation specific to the particular event of production.[309] Audit evolves around the system's abstract divisions to be able to limit its operations to only reviewing the managerial structures which in turn audit their internal divisions, which also have specific local forms of audit, and so forth, throughout the whole of the abstraction stack. Provided the audit pattern is consistent enough to be able to use a unified approach through and through, the audit is ensured to be completely scalable and will not provide any significant bottlenecks by trying to absorb excessive resources on the same abstraction layer.

Knowledge validation mechanisms in epistemic infrastructure

The key principle of an epistemology of the production circuit is that the knowledge, once it has been assimilated and organised into an infrastructure, can be effectively used to evaluate new knowledge. This principle lies at the core of most of the production systems' complexity avoidance tactic, which splits the knowledge into one which is organised and culturally integrated – explicit knowledge – from the one that is a matter of negotiation, or tacit knowledge. This tactic comes naturally as the system proceeds with codification and abstraction routines of parsimonious cognitive behaviour. As most of the other agent-based formations, including the ones composed of human, non-human or organisational entities, production systems tend to proceed in the direction of lesser effort, which means retaining and re-using previously assimilated knowledge as much as possible. Existing knowledge, ordered and made available as a resource repository saves the effort of either creating or searching for it, and therefore the activity of integration, or converting knowledge to explicit state is one of the agent's core parsimony tactics. However, it is not always employed, particularly where it is possible to accumulate knowledge in its tacit state within the enclosed agent groupings. The reason for that is that the activity of integration – for example, creating a comprehensive body of technical documentation – can appear as a considerable expenditure of time and resources. As demonstrated in Chapter 5, tacit knowledge is also open to valorisation mechanisms owing to its added effectiveness due to the large momentum in local situations. This usually means, as some of my field cases demonstrate, that the organisation is not likely to allocate the resources to creating documentation, refactoring or addressing technical debt unless there is a critical system failure or another crisis situation.[310]

---

[309] Cf. Toscano, 2008: 275, 277.

[310] See Appendix, CS6, CS21.

Future research

Since the present thesis opens up a new way of thinking about such a broad sphere of professional IT practice as DevOps, a few of the exciting research opportunities had to be left out for the sake of creating a balanced foundation for the argument as a whole. Some of the urgent next steps for developing the present study lie in the areas of software distribution, the co-implication of software and hardware production processes, the relation between momentum and traction, the valorisation of computation, as well as a more nuanced exposition of the role of affect in high-complexity production.

Let me draw some examples that link back to the argument as it appears in the thesis. In terms of distribution, as mentioned in Chapter 3, the notion of a system environment as a set of resources that encapsulates the local user interactions has to be further developed as an analytical tool to be able to unpack the meaning of user-side situated knowledge that evolves in runtime. Another potentially fruitful trajectory is the conjunction of hardware and software in terms of their production processes. For example, how would the present research approach the production of a smartphone – should it be seen as the result of the industrial mass manufacture conditioned by the software? Or if it appears simultaneously as a physical product and a software service, would it raise questions as to where the production process stops, and whether the device itself should be approached diffractively, as a result of production and the means of distribution? In terms of affect, only a cursory examination was possible within the limits of this study, as Chapter 2 explained, and a more thorough investigation is required to evaluate the involvement of affect as the means for reducing complexity. As another example, there needs to be a more detailed account of the technological system mobility from momentum to traction-based operative principle, to continue the discussion of the two notions started in relation to performativity of labour in Chapter 1. Lastly, as mentioned in the discussion of the supply chain capitalism model, there needs to be a further diffractive analysis of the roles of participants of production where their class relations overlay their organisational stakeholder roles.

To conclude, I have to note how the aim of my dissertation has evolved during the process of my research. As mentioned in the General Introduction, the research starts with the aim of finding out what prevents organisations from developing their software in an easier way, so that they fully utilise the power of technology that they have access to. Throughout the chapters, however, the problem reveals itself to be not as trivial and shifts the interest of the research towards the issues of method, making it necessary to clarify the key themes as summarised in this Conclusion. The question at the end of the thesis sharpens to ask, how could the epistemology be mobilised as a productive assemblage for planning and control in situations of exponential complexity increases? Providing any informed answer undoubtedly requires further ongoing investigation, which falls outside of the scope of the present thesis.

The main result of this study, which is a sketch of topology that traces the problem space involving the problematics from both DevOps research and software studies, comes, despite its preliminary nature, as a benefit to both disciplines. It suggests that developing a more nuanced critical inquiry into DevOps of software capitalism demands further and urgent research for several reasons. One is that complexity plays an important role in the growing interoperability of not only technical, but also human and environmental factors of contemporary capitalist juncture, and therefore the study of complexity has to be seen as a concern of the humanities scholars as much as their colleagues in the departments of computer science or management. Furthermore, as the cultural sphere sees an increase in the active involvement of more-than-human agents, their encounters require a further intensive investigation. This is evident in the recent integration of artificial intelligence-based tools in cultural production, and the growing computation-based links between the continuously deployed infrastructures of global sales and logistics platforms, on the one hand, and climate crisis and organisational ethics, on the other. In addition, there is a need to create a comprehensive exposition of complexity effects caused by depreciating computation because they are likely to have long-term effects across the production, operations and audit practices.

Appendix

The Appendix lists the case studies which I carried out both as part of my PhD research and my digital product lead employment at the organisation referred to throughout as JX. JX is an online media outlet publishing daily briefings on a variety of cultural topics, focusing on young creatives. Each study is meant as an illustration of a specific skill that I had to utilise to carry out the casework.

CS1. Planning and delivery

To illustrate my approach in this area, I can use a case study of the XY project that involved online film screenings which JX had organised during the COVID-19 lockdown period between March and October 2021 to increase the audience and visibility of the journal. Production requirements for XY included creating a new website section to host the film screenings, implementation of a streaming platform and design of the associated promotion materials.

I have approached the project as a series of the following tasks:

- To create a detailed requirements document based on stakeholder meetings.
- To do the research for the film streaming options.
- To coordinate the design and production of the XY landing pages, as well as film streaming platform integration.
- To coordinate the delivery of marketing collateral.

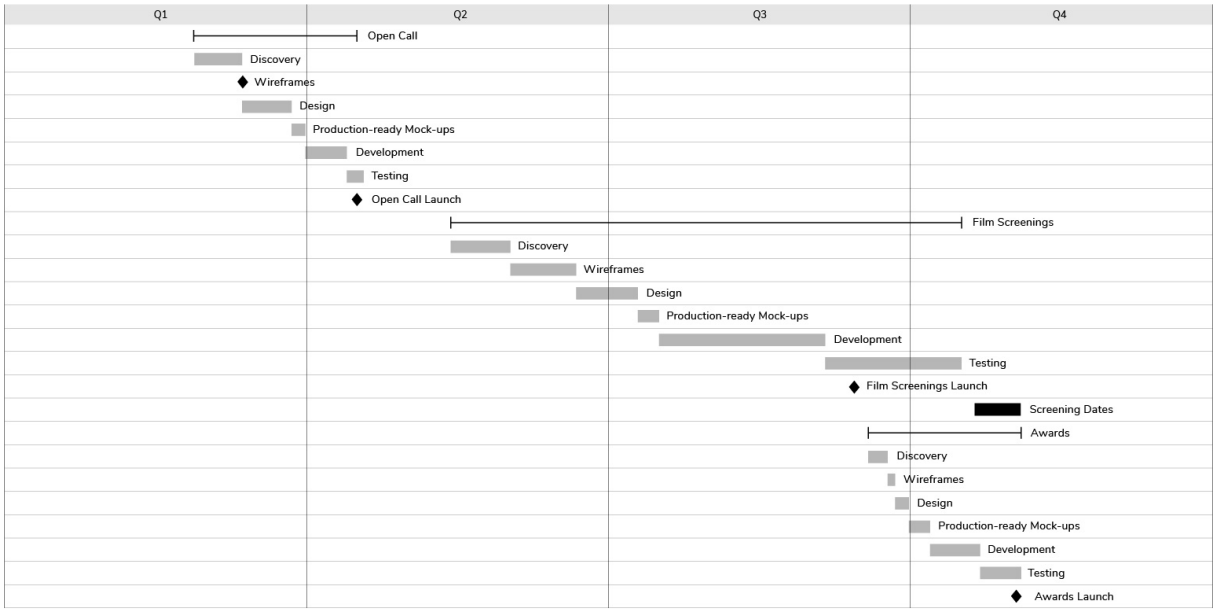

Fig. 19. Production phases and releases.

The project workgroup consisted of the producer from the editorial side, two developers and two designers. When building the roadmap, I have split the production efforts into three versions: open call, film festival and awards. (Fig. 19) These corresponded to the JX landing pages which had to be released at three points: before the festival, at festival start and when the winners were announced. I planned and coordinated the delivery phases according to the production best practice: discovery (including opportunity, planning and estimation), design (including wireframes and production-ready mock-ups), development and testing (Fig. 20). Such treatment had allowed me to effectively track the tasks pertaining to each release.

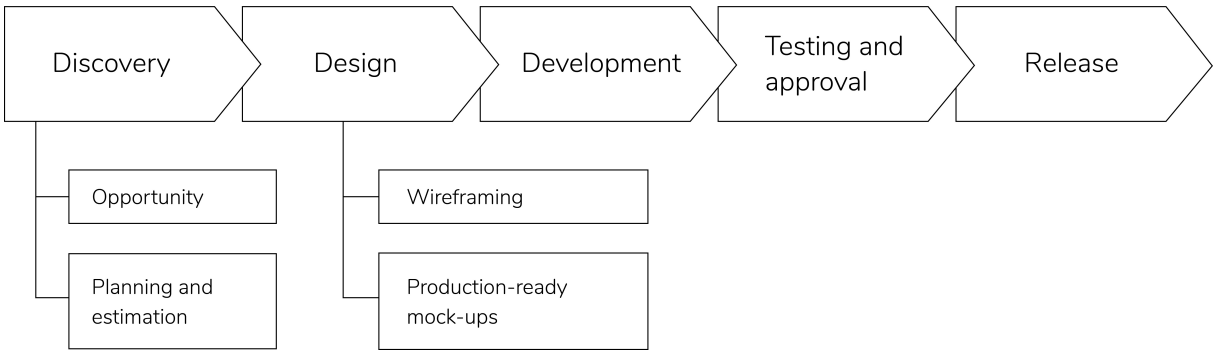

Fig. 20. The production pipeline.

CS2. The use of tools to balance priorities and mitigate the risks

In the context of the XY project as described in CS1, I have balanced the priorities and mitigated the risks with the three industry grade tools: Smartsheet, Jira and Confluence. While Smartsheet worked

well for stakeholders, Jira was ideal for development, and linking these two tools allowed me to create a fluent conversation across the different organisation strata. Lastly, Confluence was used as a technical documentation space available for all.

Smartsheet. The roadmap I created in this online Gantt suite had allowed stakeholders, producers and marketing to estimate the efforts and evaluate the risks before taking any action. For XY, I used Smartsheet to set the timeframes for the three releases, populated each with associated production, editorial and marketing activities, and assigned tasks to relevant team members. This has allowed me to report based on person, team, task type, or project stage. Where possible, I have linked the tasks to corresponding Jira issues.

Jira. As a tool for managing support tickets, Jira was a good fit for balancing priorities and dependencies in delivery of XY, once the strategy was approved on the stakeholder level.
I have used Jira for the following tasks:

- Composing **user stories** for all aspects of development, for example: 'produce video for the banner', 'create an anchor link to festival passes', 'stabilise the open call page performance'. Each user story contained the requirements, rationale, current context and any additional information.
- **Backlog grooming**: organising the backlog stories in order of their priority, based on the consensus within the organisation and the production team. Decommission or de-prioritise the stories which were no longer relevant.
- **Sprints**. Opening, closing and reporting on sprints in two-week intervals. Constructing upcoming sprints with stories from the top of the backlog, depending on team velocity and story point estimates.
- **Releases**. Tethering Jira releases to pull requests in GitHub, which the developers used for version control. Each release deployment was marked with the corresponding Jira release tag. After each release, I ran a retrospective with the XY workgroup to address any issues, celebrate the achievements and discuss the work approaches.

Confluence. Creating pages for initial requirements, meeting minutes, feedback, retrospectives, and supporting the developers in writing up the technical details, such as streaming platform integration.

CS3. Annual planning

In the first quarter of 2022 at JX, I was responsible for creating a draft high-level roadmap. This task included:

1. Gathering what was already known about the projects planned for that year, compiling draft requirement documents and creating wireframes which would show essential features.
2. Compiling the draft roadmap while discussing the collected materials with the developers, who would advise on the rough time estimates.
3. Adjusting the roadmap draft so that all team members have a steady flow of work.
4. Stakeholder presentation, paying extra care to the responses in terms of the delivery priorities and projected deadlines.
5. Revisiting the roadmap in view of the feedback and sharing across the team as the initial version of the workable plan. The emphasis here was that the plan is in no way set in stone and would be further adjusted as the projects take shape.

As a result, the team had an understanding of the composition of each project throughout the year, which allowed them to organise their work accordingly.

CS4. User research

At JX, I have conducted user research as part of the main navigation redesign project, to understand the goals and aims of such redesign. In this case, I was only tasked with surveying company employees because a brief audience survey had been done shortly before. I carried out two group surveys of staff in both company's offices, and a series of face-to-face semi-structured interviews with stakeholders. In the second stage, I have identified that these conversations were, in fact, not sufficient because the group was already too familiar with the product, and a more thorough user testing and more audience surveys were required. Knowing that JX lacked the resources to carry out such testing, I organised a meeting of JX staff and a third-party business analyst consultancy, who came up with a proposal for creating a viable user research programme. The approval of the initiative, however, was delayed for reasons outside of my control, and given the project deadlines, we had to complete the initial version of the new main navigation based solely on the audit results already at hand (see CS7 and CS12 for more details). Given such limitations, we could still argue the successful translation of user needs into tangible outcomes, since we have created a search function and a sidebar menu where none of those existed before. Going forward, I had persisted in advocating a continuous user research protocol, to be able to better instruct further UX improvements.

CS5. Discussions with technical teams

I structured communication within the production unit at JX around the weekly 30–40 minute meetings, during which each team player reported on current progress. The weekly rather than daily cadence made sense because most team members were working part-time. Moreover, some production team members also attended the daily editorial stand-ups, and also had a chance to communicate

about particular projects in project-specific meetings. My tasks included hosting of meetings, creating the backlog of Jira tickets, updating the roadmap and articulating the pain points for further discussion beyond our team. I was also responsible for deciding on the optimal approach to the scope of the upcoming week in terms of the task implications and trade-offs of maintenance vis-à-vis new feature development. Such an approach to team discussions contributed to the confidence of each of our colleagues in their work and aided developing a sense of trust within the team as a whole.

CS6. Reliability and security

Following a website performance failure due to a suspected attack in October 2021, I had decided to start an initiative to improve the platform security. I had then obtained approval to hire a freelance DevOps specialist, with whom we have formed an intensive working relationship over the period of the following five months. Together, we have implemented a range of security measures across the several key areas:

- Content delivery network (**Cloudflare CDN**): updates to security settings, firewall and error pages.
- Amazon Web Services infrastructure hosting (**AWS**): updates to users, groups and permissions.
- AWS: scaling down the infrastructure to optimise the costs.
- AWS: create a proposal and cost estimates for high availability architecture and auto-scaling to increase product reliability.
- JX **content management system**: drafting the password rotation policy, updating users, implementing captcha.
- **Google Workspace**: updating user groups and users, two-factor authentication, security updates on user devices.
- **Physical server**: migration of data to the cloud storage (Google Drive). The migration was supplemented by creating the automated backup suite, briefing the staff on the new Drive usage and decommissioning of old equipment.
- **Hardware support**: revising the annual rolling contract with hardware support company to understand what we are paying for.

The benefits of the initiative were as follows:

- AWS optimisation brought a 65% reduction in monthly costs without any losses in service quality.
- Using Drive and backups improved compliance with company data storage policy.

- More detailed analytics on Cloudflare; the rest of its optimisation required more time before verifying the results.
- Captcha reduced the number of failed login attempts to the company's services (no specific KPIs), which suggested previous malicious attacks.
- Hardware support: we have found that we no longer required third-party support services because the nature of JX operations had changed too much over the years – no physical server, no physical location, etc. This meant further reduction in support costs.

CS7. UX

My approach does not include the formal handling of UX, and up to this point has been lightweight, which can be demonstrated through the following example. In the navigation redesign case mentioned in CS4, JX has worked with an external senior design consultant. The consultant provided the static designs, videos that demonstrated animations and the key effects, along with the interactive prototypes. My role was, often together with the project manager, to test the prototypes on the range of devices, to make sure that the page transitions correspond to the approved user journeys, get the client approvals and to give feedback to the designer. While this approach to UX was culturally appropriate to JX, I'm interested in further developing my command of formal UX techniques.

CS8. Analytics

My job duties up to now did not include the in-depth data analytics as such, and was limited to using the two tools:

AWS. I had created dashboards for CPU and memory load balancing trends to understand the capacity demands, had been monitoring the alarms and accessed the Cost Explorer to report on the details of monthly billing. Overall, I have a basic familiarity with AWS reports.

Cloudflare. I mainly reviewed the dashboards for audience statistics, however my knowledge of this platform so far is limited.

CS9. Quality Assurance (QA) and User Acceptance Testing (UAT)

In the JX migration process to the new version of the platform in 2018–2019, I was a primary point of contact to the third-party IT suppliers who were commissioned to carry out the technical parts of the job. In this project, I was involved with QA and UAT as follows:

QA. I assisted with managing the backlog of support tickets and handled the communication between JX and external quality assurance contractors around the issues that required further details from JX.

UAT. I approached the UAT process for each round of testing in the following way: once the new release was ready for the UAT, it would be made available on the staging environment. I would then confirm the testing session times with the dedicated members of internal team at JX (the UAT team). Next, they would log in and leave their comments in the shared spreadsheet. For more intensive sessions, we sometimes found it easier to schedule a conference call, during which the UAT team would test the product, and I would fill out the sheet. During the sessions, I also made sure that we tested from different geographical locations and on different devices, using the testing emulation tools such as Lambda where needed. After completing each round of feedback, I would have a call with the contractors carrying out the work to discuss the test results.

CS10. Technical documentation

High-level documentation. During the JX platform migration to AWS as the new cloud infrastructure in April 2020, I was, among other things, responsible for delivery of the project's technical documentation. I had personally authored the high-level documentation pages, while the developers and DevOps were documenting the corresponding technical parts they were working on. This had resulted in a section of the company's Confluence wiki, detailing various aspects of migration and implementation of Continuous Integration: ELK stack creation, Jenkins pipelines, Git workflow, AWS environment schemas, how-to for Nginx and PHP configuration, and more.

Feature specifications. A case of ElasticSearch implementation, discussed in CS12, is an example of me writing the feature specifications suitable for technical implementation. For the Search, after the initial part of the work, including the discovery and visual concept was done, I had prepared a draft of technical specifications and coordinated the follow-up discussions, approvals and the refinement of the specs list. As a result, I would be able to produce a roadmap and the cost estimates, which would allow me to create a viable project proposal stakeholder presentation.

Data to knowledge collection flow. The ongoing method for collecting and integrating the project data has consisted of a few key stages, as demonstrated in Fig. 21. First, taking notes in meetings and following up each meeting with an email stating the date, attendees and key takeaways. Second, to create tickets based on the work items identified in the notes. The main purpose of the ticket is to generate a list of *requirements*. It has a list of *decisions* which were made and a list of *actions* to take, where each action is linked to a specific individual to follow up with. Decisions, actions and requirements are updated from meeting to meeting as the work progresses. Third, after each release or other milestone, we run a *retrospective* that allows us to identify failures as well as successes. In the fourth step, after the retrospective, all data is transferred from Jira to a more permanent storage, where it is systematised in a

way that would be easier to access later or by individuals who were not involved in the work previously. Lastly, based on the resulting knowledge base, I submit an annual report at the end of each year.

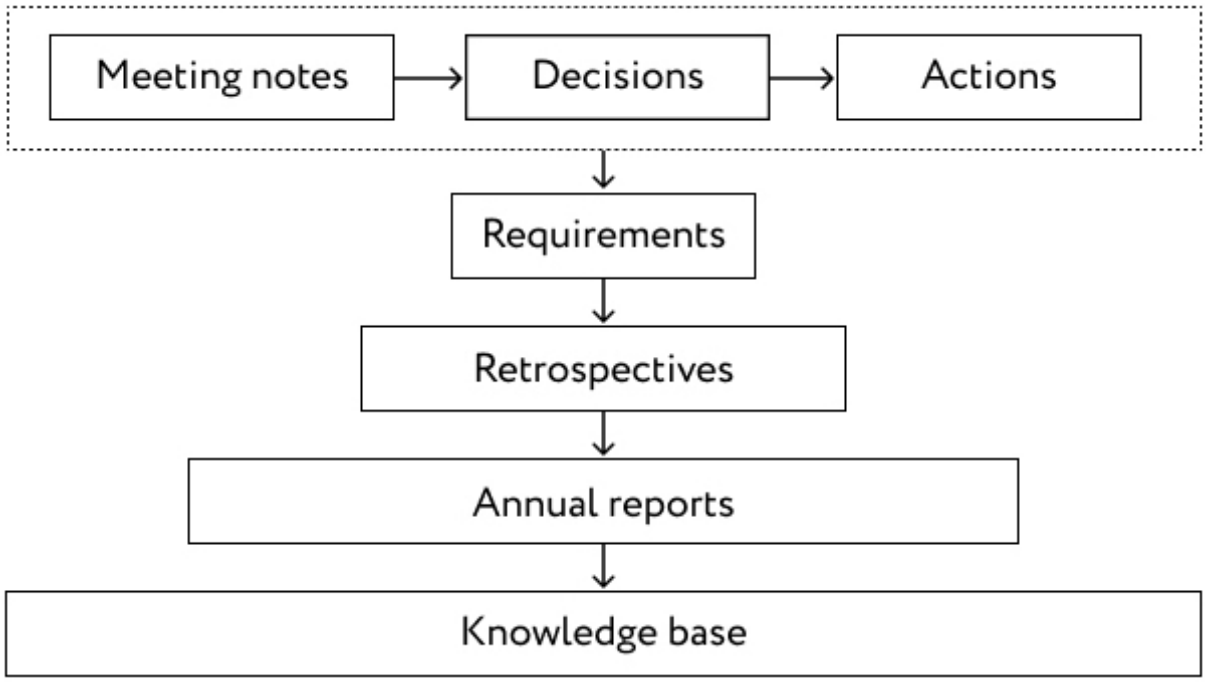

Fig. 21. Data collection flow.

CS11. Stakeholder communication

When delivering the security suite described in CS6, I was reporting the progress in weekly management meetings to the business owner, creative director, editor-in-chief, head of operations and head of marketing. Here, the low degree of technical familiarity among the executive staff was a challenge that I addressed by focusing not on the technology itself, but on budget and time considerations. I would avoid using any technical jargon. As an example, for the old and failing server problem, I had presented a few different solutions and their associated costs. One option would be to replace the old server with a new box, the other – less costly and more secure – would be to decommission and migrate the data to cloud storage. This made it easy for the stakeholders to evaluate the trade-offs and take the decision quickly.

CS12. Initiative

I have proactively identified the JX website search feature as underperforming in a few respects, both search quality and UX, and came up with the updated requirements that included a more thorough search plug-in (Elastic Search), a prominent search field in the top navigation and the ability to filter search results by the category, location and content type. This proposal was supported by marketing,

who saw this initiative as an opportunity to increase the visibility of the JX's rich archive database. The staff designer was also inspired by the project and came up with a set of visual concepts, adding such ideas as the visual search pop-up box and the results counters. Next, I had split the search delivery into phases (Fig. 22), and have found an opportunity to hire a back-end engineer for the implementation of Elastic Search plug-in after obtaining the approvals on the new navigation designs. As a result, despite the missing protocol for KPI tracking, business owners had evaluated the overall accessibility of the website content as improved and marketing had confirmed the boost in targeted promotions.

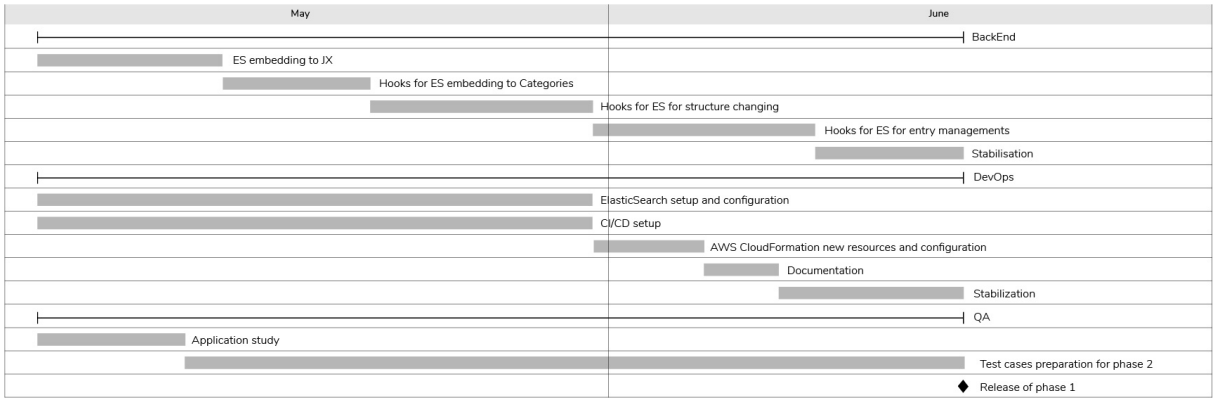

Fig. 22. A proposed roadmap for Elastic Search Phase 1.

CS13. Governance approvals

In my role at JX, I was not directly involved in governance approvals, and due to the small size of JX itself, no navigation of complex structure was necessary. However, such negotiations is a direction that I see myself shifting towards in the coming years.

CS14. Influencing

The 2018 YY project at JX presented a case where I used my influencing abilities to address the silos issue. The project involved production of several short films, which meant that key members of the team had to be away on site making the footage. However, the silo effect meant that little of the project information had seeped into the production team, and we were faced with the dilemma of creating an online presentation of the films on a short deadline and with no prior planning. Without having a direct authority to change the silo situation, I was, however, able to influence the wider company culture over a period of time so that as we went along, the practice of using shared roadmaps and collective discussions and planning of releases became a part of the usual approach. The tangible result were the new weekly management meetings, which provided the opportunity for heads of all departments to communicate and report on their progress.

CS15. The archive project

This case study stands apart from other activities in my fieldwork because it was conducted as a separate project after my empirical research was finished, and in fact after the official closure of JX as the media channel in 2022, due to the geopolitical tensions between Russia and Ukraine, which were the main sites of the journal's content. The core stakeholder requirement for the project was to transform JX to a digital archive, with an aim to preserve its legacy. As a former product lead for JX, I was involved with the delivery of the archive with the new team, trying to bridge some of the major gaps in knowledge to maintain the required traction. At the start of the project, I came up with the project proposal that included a user story in the following form:

> *As [an academic institution user] interested in the history and culture of [this geopolitical region],*
> *[I want to] have a versatile database search tool*
> *[so that I can] easily find the collection of materials on the topic I'm interested in.*

The user story formulated as such has largely instructed the project requirements, such as easy accessibility of required content and the three main user activities of searching, sorting and browsing the archive entries. To facilitate them, I proposed to add three new features to the existing website. First, a new home page that would no longer have recent publications and instead offer a prominent search field. Second, the expanded Advanced Search landing page that would feature filtering by date and tag in addition to existing category, location and content search.[311] Lastly, the new Catalogue page, which would act as the website's table of contents and list all the JX categories, locations and tags. Following from the proposal, I have created a product map (Fig. 23) that would help explain how these new features would sit within the existing JX structure. On the map, the pages would be functionally divided into Sort pages (Home page, Advanced search and Catalogue), Browse pages (such as Travel, Photography and other Category pages) and Content pages containing the actual editorial content.

---

[311] See CS12 for more details on original Elastic Search work.

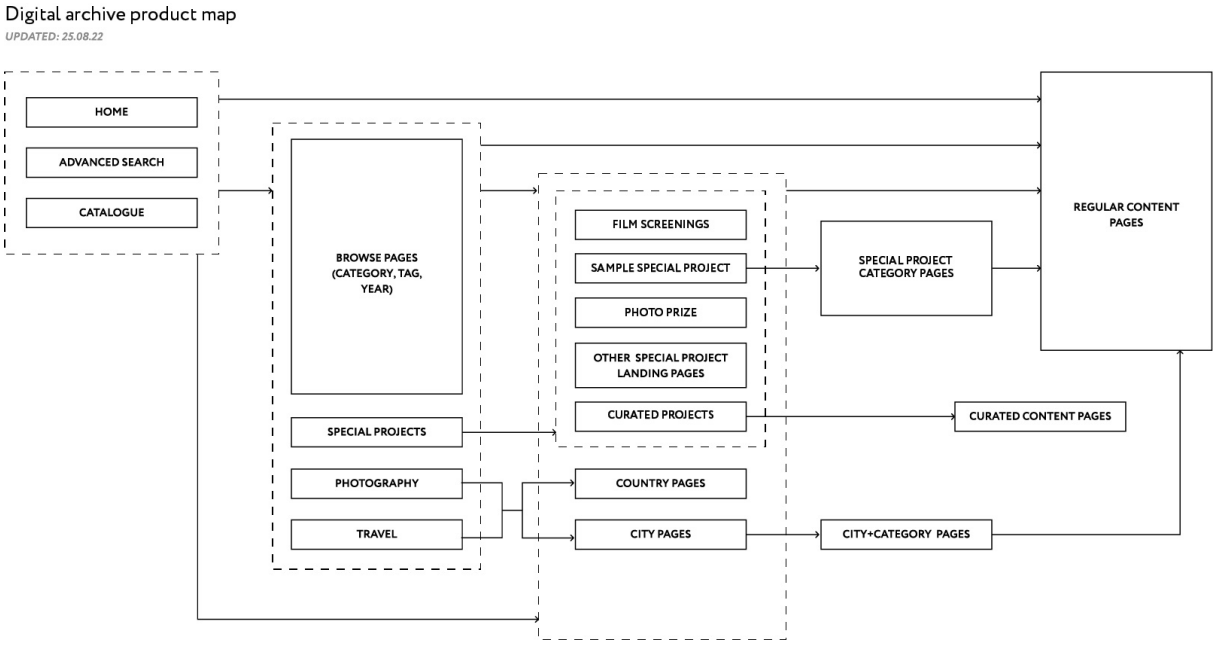

Fig. 23. Digital archive product map.

After presenting the work scope to the stakeholders and the discussion about the timeline and budgets, it was possible to come up with the project work breakdown structure in the form of the roadmap (Fig. 24). The roadmap would feature the two main types of work to carry out, design and development, and have split the production efforts per feature in terms of their dependencies. For example, the home page and updates to the navigation had to be done prior to the work on the Elastic Search or the Catalogue. At that point, stakeholders had also agreed to deliver the work in two phases so that we could launch and test the new home page and navigation as a first step, and deliver the rest of the work in the second step. It is important to note that this roadmap was only created as the initial indication of which work was required, and was subsequently used by the IT suppliers who were assigned to do the work, to create their own project documents, estimates and to allocate resources.

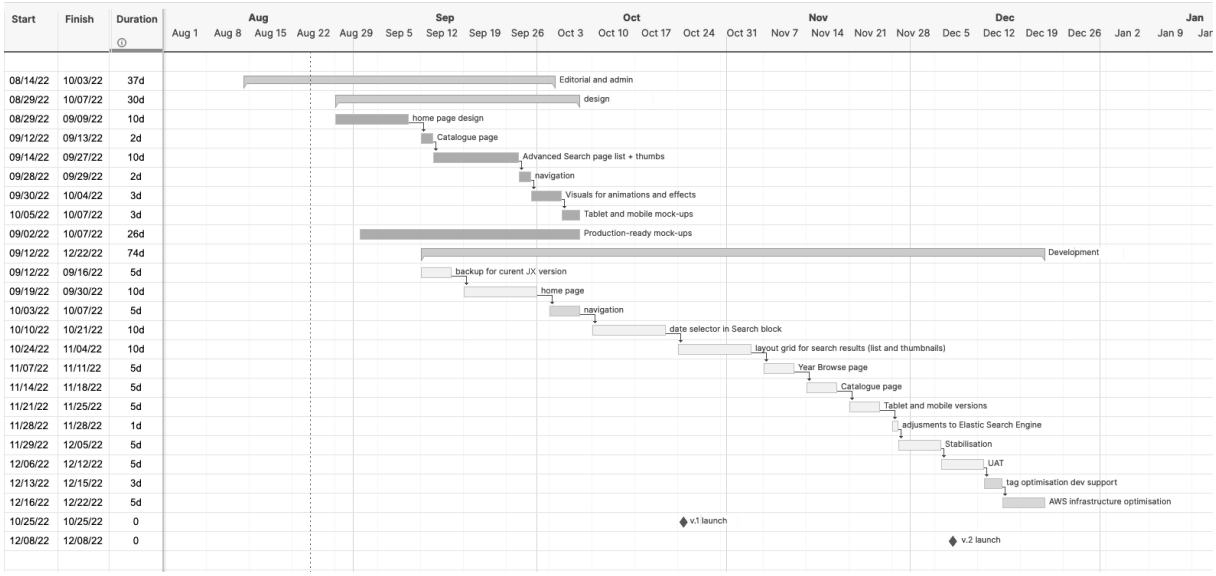

Fig. 24. The archive project roadmap in the proposal phase.

Production-wise, the key problem on this project was that we no longer had the previous team members, the back-end and two front-end developers, which led to the inevitable loss of momentum. This meant that the back-end engineer and webmaster freshly enlisted on the project could not simply start building where the old production team left off, even though we had ensured that all the required system access and product documentation were provided. Some of the features, such as top navigation, had to be re-created from the ground up, and some of the content work that seemed easy to do, such as re-assigning the tags and categories, had required extensive investigation and in the end had to be excluded from the project scope due to required extensive redesign of the product database architecture. The main outcome of the case study was that despite my concerns, the loss of momentum did not seriously damage the overall team performance. The onboarding was quick and seamless, and even though the new features created regression problems in places where they were not compatible with the existing code base, most of them were addressed within the project scope.

CS16. Customer experience

At JX, the situation of issue reporting, prioritisation and troubleshooting can be illustrated through the case of so-called 'broken pages'. After the migration to the new version of the platform was completed in the early stage of my employment at JX, we discovered that dozens of old pages had manually adjusted code which meant that the layouts appeared distorted, and there was no automated way of fixing them. My task was to establish a routine that would enable the support staff to deal with the problem in a systematic, longitudinal manner. I have come up with the following protocol: the editorial team, who did not work in Jira, would log in the faulty URL and the description of the issue in the spreadsheet shared online. From there, developers would address the small fixes, and I would write up

the stories for larger ones – for example, where the slideshow gallery plug-in had to be changed for an entirely new one. This way we covered a lot of ground in a short amount of time. After we had dealt with the urgent matters, I moved on to address a larger issue – that the 'broken pages' were deployed into production in the first place. This concern resulted in the implementation of the three environments: development, testing and production, which allowed capturing most bugs before publishing to the live version of the website.

CS17. Workshops

For identifying opportunities, I primarily analysed the competition and the user research data available, such as described in CS4 and CS12. I have also delivered the workshops whenever a new tool or practice was introduced in our workflow. For example, as the team was familiarising itself with the new Continuous Delivery workflow, I have conducted a workshop on story mapping, a software production method discussed in Chapter 4. This included creating a shared online whiteboard, a tool simple enough to use by technical and non-technical staff. Once everyone was logged in, I would introduce the concept of thinking about new features in terms of storytelling and the process of splitting complex scenarios into releases. This explanation was followed by the practical part, during which the participants would collectively create a story map of their morning routine, with different scenarios. The outcome of the workshop had an effect that had a vast resonance in the team – people had a great grasp of the concept of versioning after the workshop, and referred to the event later as a useful learning experience.

CS18. Advocating change

One case of a 'disagree and commit' situation I had encountered was during the security and reliability initiative as described in CS6. As part of the initiative, our research had shown that the organisation had used considerably more CPU and memory than was needed because the generic infrastructure settings of the initial installation were not revisited and adjusted regularly. I have come up with a proposal to optimise the infrastructure for present requirements. My proposal, however, was initially rejected, due to the stakeholder's engagement in other aspects of the business. I have met this decision with an open mind, in the hope that a chance to come back to the proposal will present itself. And indeed, some time later, the business priorities have changed, and we had a new brief to reduce the costs, which allowed us to proceed with the earlier proposal. This resulted in a 65% AWS cost reduction.

CS19. Backing up ideas with data

The data I had used to back up my arguments at JX usually concerned budgets and time required to complete the work (as seen in CS11). In the case of XY as described in CS1, my task was to present such data to editorial and marketing so that they could advise on which streaming platform to use. The

objectives included ability for ticket sales, setting up the film streaming times and accessibility of audience metrics. To achieve this, I have done a comparison of costs, between a bespoke streaming platform vis-à-vis the out-of-the-box solutions. After a cursory investigation revealed that renting a ready-made platform would incur considerably fewer costs, I conducted further comparison among the third-party vendors, attending their demos and assessing the compatibility of their offers to our requirements. This process resulted with a decision on a specific service, backed up by the appropriate budget and functionality, which had fully supported the organisations' strategic objectives.

CS20. JX server migration

One such case of the proposal backed by the data had been the JX physical server migration, which was undertaken in November and December 2021 and was carried out in the fashion close to the topological case study explained in this chapter. The data and the organisation's strategic objectives came together during the server downtime incident in the last week of October 2021, which was followed by the complaint from the JX staff that the IT maintenance contractors were late in responding to JX's queries and neglected their contractual obligations, which resulted in the server downtime that lasted for over five days, even though the power blackout itself that caused the downtime was only a few hours in duration. While the client requested to simply seek other suppliers, my first instinct was to collect and organise the data by reconstructing the incident and examine the current IT support contract to draft a more precise list of requirements. Without doing this, I wouldn't have been able to tell which services we needed to seek to replace, and whether we required those services at all. In addition, the original complaint was raised after the incident was over, which meant that I was not aware of all the details and had to begin by creating a timeline that would reflect the sequence of events that caused the complaint. This was done by examining the emails and conducting the three brief interviews with the finance manager, the operations manager who were engaged in the conversation with IT contractors and the client's web developer, who had travelled to the JX offices to switch the equipment off and back on. At the end of this phase, I had the following Gantt chart (Fig. 25).

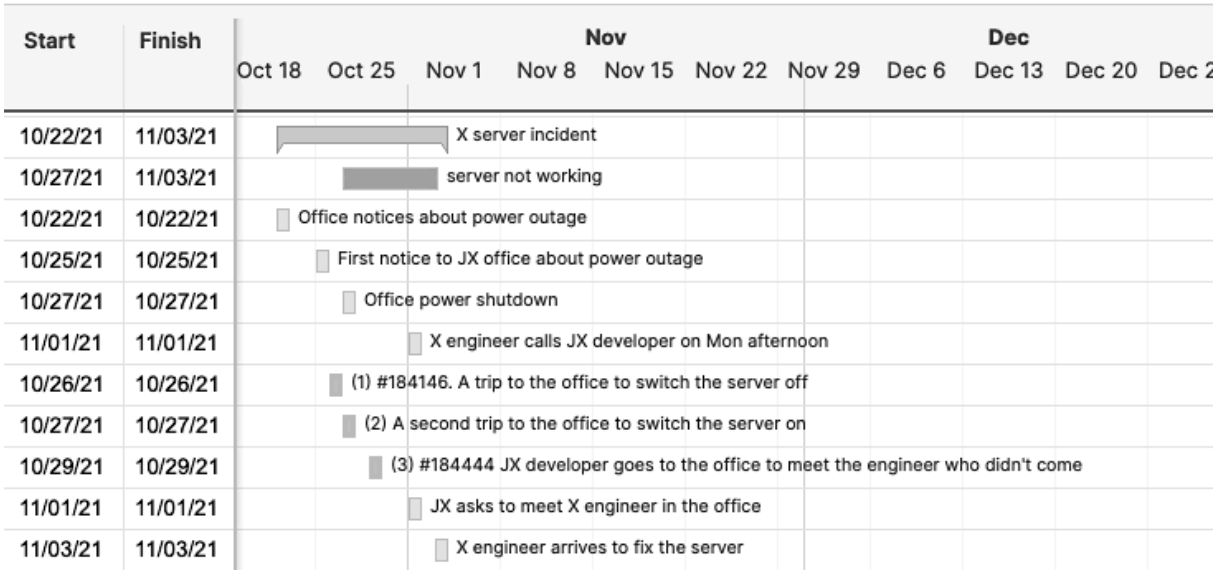

Fig. 25. The first reconstruction of the server incident.

This, however, only led me to believe that there was no fault of the suppliers, but instead the server downtime had lasted for five days for the reason that no JX staff was available in specific moments, and the communication was handled by different people, which led to inevitable delays. In addition, I have learnt from the contract that we used IT services of three different kinds: server backups and maintenance, software support and hardware support.

As the next step, which signalled the beginning of phase 2 of the case study research method (case-based work), had a planning session with the DevOps specialist, during which it became clear that the problem was not in the choice of a particular supplier. Rather, it was in the performance of the server itself, which was low because it had been in operation for a long time and needed a replacement. Instead, we then proposed to migrate the server online, and create a Team Drive on Google, a web service that members of JX staff were already familiar with, and thus would be happy to switch to. This solution also solved the security problem, since it meant that staff would no longer seek to save their files in an unregulated way, and have all the files stored via the company's assets. This solution was welcomed by the JX staff, which had allowed the DevOps and me to proceed. The following steps were included, as shown in Fig. 26 alongside the reconstruction of the initial incident: a trip to the office and sorting out the access rights; remote server access, creating the required policies: cloud storage, backup policy and the new folder hierarchy. Lastly, we had to create an exit strategy from the current IT support we currently had, which was the original requirement, however now we saw that there might be a business risk in cancelling the contract with them completely, due to the possible unpredictable circumstances that no-one could deal with apart from them.

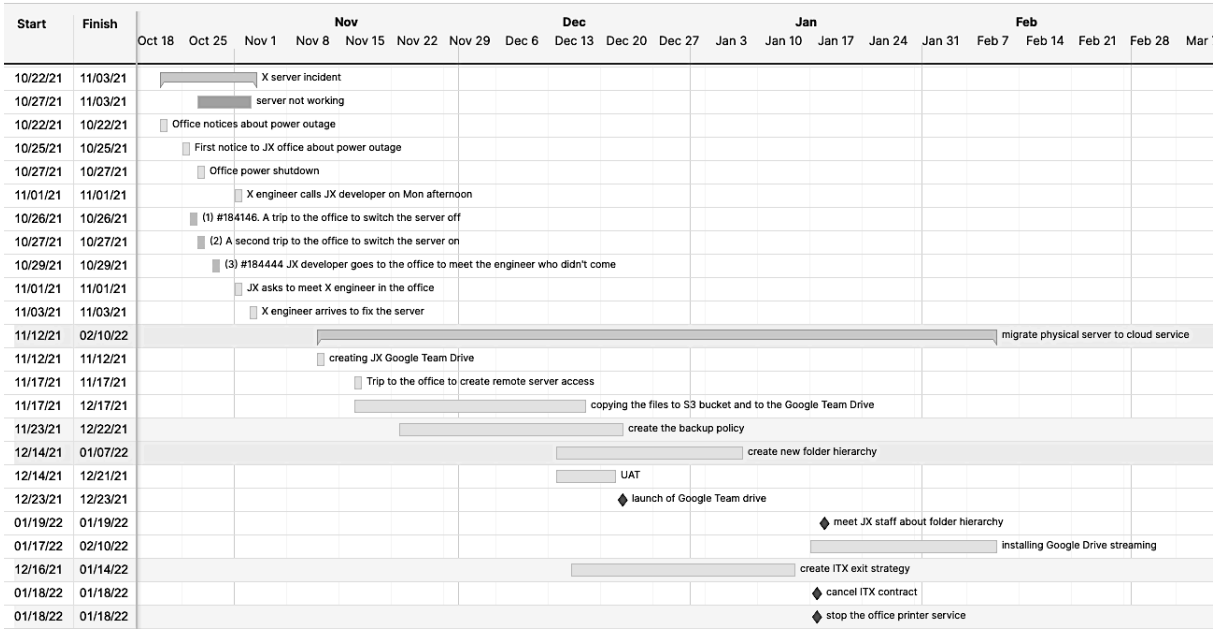

Fig. 26. JX physical server migration.

At this point, we could propose to amend the agreement, and take out the clauses that covered the server maintenance and backups, since that was now moved to the cloud service, and we no longer needed any software and hardware maintenance, since this was covered by the respective manufacturers. Thus, the project reached its closing stage, which resulted in the following activities: workshops with JX staff on using the new cloud-hosted server file system, and the proposal for new ad hoc IT maintenance. In terms of topological continuities, we could observe that a more transparent relation was now in place between the business and the organisational planes of the technological system, since the role of the IT contractor was clarified and the unnecessary step of storing the files in the resource that required additional maintenance and resources was eliminated.

Bibliography

Allen, Peter, Steve Maguire, and Bill McKelvey, eds. 2011. *The SAGE Handbook of Complexity and Management*. London: SAGE.

Allspaw, John. 2013. "Kitchen Soap – Counterfactual Thinking, Rules, and The Knight Capital Accident." Kitchen Soap. October 29, 2013. https://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/.

Arthur, W. Brian. 2009. *The Nature of Technology: What It Is and How It Evolves*. London: Free Press.

———. 2021. "Economics in Nouns and Verbs." *Journal of Economic Behavior & Organization* 205 (April): pp. 638–47. https://doi.org/10.1016/j.jebo.2022.10.036.

Azhar, Azeem. 2021. *Exponential: How Accelerating Technology Is Leaving Us Behind and What to Do About It*. London: Random House Business.

Barad, Karen. 1996. "Meeting the Universe Halfway: Realism and Social Constructivism without Contradiction." In *Feminism, Science, and the Philosophy of Science*, edited by Lynn Hankinson Nelson and Jack Nelson, pp. 161–94. Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-009-1742-2_9.

———. 2003. "Posthumanist Performativity: Toward an Understanding of How Matter Comes to Matter." *Signs* 28 (3): pp. 801–31. https://doi.org/10.1086/345321.

———. 2007. *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*. Durham: Duke University Press.

———. 2010. "Quantum Entanglements and Hauntological Relations of Inheritance: Dis/Continuities, SpaceTime Enfoldings, and Justice-to-Come." *Derrida Today* 3 (2): pp. 240–68. https://doi.org/10.3366/E1754850010000813.

———. 2011. "Nature's Queer Performativity." *Qui Parle* 19 (2): pp. 121–58. https://doi.org/10.5250/quiparle.19.2.0121.

———. 2014. "Diffracting Diffraction: Cutting Together-Apart." *Parallax* 20 (3): pp. 168–87. https://doi.org/10.1080/13534645.2014.927623.

———. 2015. "TransMaterialities: Trans*/Matter/Realities and Queer Political Imaginings." *GLQ: A Journal of Lesbian and Gay Studies* 21 (2–3): pp. 387–422. https://doi.org/10.1215/10642684-2843239.

Bateson, Gregory. (1972) 1987. *Steps to an Ecology of Mind: Collected Essays in Anthropology, Psychiatry, Evolution, and Epistemology*. Northvale, N.J: Aronson.

Beniger, James Ralph. 1986. *The Control Revolution: Technological and Economic Origins of the Information Society*. Harvard University Press.

Bijker, Wiebe E., Thomas Parke Hughes, and Trevor Pinch, eds. (1987) 2012. *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*. Anniversary ed. Cambridge, Massachusets: MIT Press.

Boisot, Max, Ian C. MacMillan, and Kyeong Seok Han. 2007. *Explorations in Information Space: Knowledge, Agents, and Organization*. Oxford ; New York: Oxford University Press.

Brooks, Frederick P. (1975) 1995. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary ed. Reading, Massachusets: Addison-Wesley Pub. Co.

Brooks, Rodney. 2021. "A Quadrillion Mainframes on Your Lap." IEEE Spectrum. December 21, 2021. https://spectrum.ieee.org/ibm-mainframe.

Brown, John Seely, and Paul Duguid. 2000. *The Social Life of Information*. Boston: Harvard Business School Press.

Bush, Vannevar. 1945. "As We May Think." *The Atlantic Monthly*, July, pp. 101–8. https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/.

Byrne, D. S., and Charles C. Ragin, eds. 2009. *The SAGE Handbook of Case-Based Methods*. Los Angeles ; London: SAGE.

Campbell-Kelly, Martin, William Aspray, Nathan Ensmenger, Jeffrey R. Yost, and William Aspray. 2014. *Computer: A History of the Information Machine*. Third edition. Boulder, CO: Westview Press, A Member of the Perseus Books Group.

Castells, Manuel. 2010. *The Rise of the Network Society*. 2nd ed. Vol. 1. Chichester, West Sussex; Malden, MA: Wiley-Blackwell.

Clough, Patricia T. 2008. "The Affective Turn: Political Economy, Biomedia and Bodies." *Theory, Culture & Society* 25 (1): pp. 1–22. https://doi.org/10.1177/0263276407085156.

———, ed. 2018. *The User Unconscious: On Affect, Media, and Measure*. Minneapolis: University of Minnesota Press.

Conway, Melvin E. 1968. "How Do Committees Invent?" *Datamation*, April, pp. 28–31. http://www.melconway.com/research/committees.html.

Deleuze, Gilles, and Félix Guattari. 1983. *Anti-Oedipus: Capitalism and Schizophrenia*. Minneapolis: University of Minnesota Press.

Dolphijn, Rick, and Iris van der Tuin. 2012. *New Materialism: Interviews & Cartographies*. Open Humanities Press. https://doi.org/10.3998/ohp.11515701.0001.001.

Drucker, Peter F. 1993. *Post-Capitalist Society*. Oxford: Butterworth Heinemann.

Eghbal, Nadia. 2016. "Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure." https://www.fordfoundation.org/work/learning/research-reports/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure/.

Engelbart, Doug. 1994. “Interview with Douglas Engelbart: Transcript of a Video History Interview with Mr. Doug Engelbart.” Division of Computers, Information, & Society National Museum of American History, Smithsonian Institution. 1994. https://americanhistory.si.edu/comphist/englebar.htm.

Èrdi, Peter. 2008. *Complexity Explained*. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-35778-0.

Fuller, Matthew, and Andrew Goffey. 2012. *Evil Media*. Cambridge, Massachusets: MIT Press.

Gad, Christopher, and Casper Bruun Jensen. 2010. “On the Consequences of Post-ANT.” *Science, Technology, & Human Values* 35 (1): pp. 55–80. https://doi.org/10.1177/0162243908329567.

Gonzalez, Maya Andrea, and Jeanne Neton. 2013. “The Logic of Gender.” In *Endnotes 3: Gender, Race, Class and Other Misfortunes*, 3: pp. 56–90. Endnotes. https://endnotes.org.uk/articles/the-logic-of-gender.

Hanlon, Gerard. 2016. *The Dark Side of Management: A Secret History of Management Theory*. London; New York: Routledge, Taylor & Francis Group.

Hughes, Thomas Parke. 2000. *Rescuing Prometheus*. New York, NY: Vintage Books.

Humble, Jez, and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley.

Hutchins, Edwin. 2000. *Cognition in the Wild*. Cambridge, Massachusetts: MIT Press.

Kim, Gene, Patrick Debois, John Willis, Jez Humble, and John Allspaw. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. First edition. Portland, OR: IT Revolution Press, LLC.

Knorr-Cetina, Karin. 1999. *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge, Massachusets: Harvard University Press.

———. 2007. “Culture in Global Knowledge Societies: Knowledge Cultures and Epistemic Cultures” 32 (4): pp. 361–75. https://doi.org/10.1179/030801807X163571.

Law, John, and John Hassard, eds. 1999. *Actor Network Theory and After*. Oxford [England]; Malden, MA: Blackwell/Sociological Review.

Law, John, and Annemarie Mol, eds. 2002. *Complexities: Social Studies of Knowledge Practices*. Durham: Duke University Press.

Lury, Celia. 2021. *Problem Spaces: How and Why Methodology Matters*. Cambridge, UK; Medford, MA: Polity Press.

Lury, Celia, Rachel Fensham, Alexandra Heller-Nicholas, Sybille Lammes, Angela Last, Mike Michael, and Emma Uprichard, eds. 2018. *Routledge Handbook of Interdisciplinary Research Methods*. 1st ed. Routledge. https://doi.org/10.4324/9781315714523.

Lury, Celia, Luciana Parisi, and Tiziana Terranova. 2012. "Introduction: The Becoming Topological of Culture." *Theory, Culture & Society* 29 (4–5): pp. 3–35. https://doi.org/10.1177/0263276412454552.

Magnani, Lorenzo. 2009. *Abductive Cognition: The Epistemological and Eco-Cognitive Dimensions of Hypothetical Reasoning*. Vol. 3. Cognitive Systems Monographs. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-03631-6.

Malone, Thomas W., and Kevin Crowston. 1994. "The Interdisciplinary Study of Coordination." *ACM Computing Surveys* 26 (1): pp. 87–119. https://doi.org/10.1145/174666.174668.

Marx, Karl. (1867) 1990. *Capital: A Critique of Political Economy*. Translated by Ben Fowkes. Vol. 1. London: Penguin Books.

———. (1885) 1992. *Capital: A Critique of Political Economy*. Vol. 2. London: Penguin books.

———. (1939) 1993. *Grundrisse: Foundations of the Critique of Political Economy*. London: Penguin books.

Massumi, Brian. 2002. *Parables for the Virtual: Movement, Affect, Sensation*. Durham, NC: Duke University Press.

Mezzadra, Sandro, and Brett Neilson. 2019. *The Politics of Operations: Excavating Contemporary Capitalism*. Durham: Duke University Press.

Mirowski, Philip. 2002. *Machine Dreams: Economics Becomes a Cyborg Science*. Cambridge; New York: Cambridge University Press.

Mitchell, Melanie. 2009. *Complexity: A Guided Tour*. Oxford, England: Oxford University Press.

Mumford, Lewis. 1970. *The Pentagon of Power*. Vol. 2. The Myth of the Machine. New York: Harcourt Brace Jovanovich.

Nagle, Frank, Jessica Wilkerson, James Dana, and Jennifer L Hoffman. 2022. "Preliminary Report and Census II of Open Source Software," 58. https://lish.harvard.edu/publications/census-ii-free-and-open-source-software-%E2%80%94-application-libraries.

Nygard, Michael T. 2007. *Release It! Design and Deploy Production-Ready Software*. Raleigh, N.C: Pragmatic Bookshelf.

Peirce, Charles Sanders. (1940) 1955. *Philosophical Writings of Peirce*. Edited by Justus Buchler. New York: Dover Publications, Inc.

Polanyi, Michael. 1983. *The Tacit Dimension*. Gloucester, Massachusets: Peter Smith.

Power, Michael. 1999. *The Audit Society*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780198296034.001.0001.

Randell, Brian. 1996. "NATO Software Engineering Conference 1968." 1996. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html.

Rubin, Kenneth S. 2012. *Essential Scrum. A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional.

Simon, Herbert A. (1947) 1997. *Administrative Behavior*. New York: Free Press.

———. (1969) 2019. *The Sciences of the Artificial*. Cambridge, Massachusetts: MIT Press.

Skelton, Matthew, and Manuel Pais. 2019. *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. First edition. Portland, OR: IT Revolution.

Smith, Adam. (1776) 1976. *An Inquiry into the Nature and Causes of the Wealth of Nations*. Chicago: University of Chicago Press.

Spolsky, Joel. 2000. "Things You Should Never Do, Part I." Joel on Software. April 6, 2000. https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/.

Star, Susan Leigh. 1999. "The Ethnography of Infrastructure." *American Behavioral Scientist* 43 (3): pp. 377–91. https://doi.org/10.1177/00027649921955326.

Sweller, John. 1988. "Cognitive Load During Problem Solving: Effects on Learning." *Cognitive Science* 12 (2): pp. 257–85. https://doi.org/10.1207/s15516709cog1202_4.

Thomke, Stefan, and Donald Reinertsen. 2012. "Six Myths of Product Development." *Harvard Business Review* 90 (5): pp. 84–94. https://hbr.org/2012/05/six-myths-of-product-development.

Thrift, Nigel J. 2005. *Knowing Capitalism*. London: SAGE Publications.

———. 2008. *Non-Representational Theory: Space, Politics, Affect*. London: Routledge.

Toscano, Alberto. 2008. "The Open Secret of Real Abstraction." *Rethinking Marxism* 20 (2): pp. 273–87. https://doi.org/10.1080/08935690801917304.

Tsing, Anna. 2009. "Supply Chains and the Human Condition." *Rethinking Marxism* 21 (2): pp. 148–76. https://doi.org/10.1080/08935690902743088.

Virno, Paolo. (2003) 2004. *A Grammar of the Multitude: For an Analysis of Contemporary Forms of Life*. Translated by Isabella Bertoletti, James Cascaito, and Andrea Casson. Los Angeles and New York: Semiotext(e).

Wenger, Étienne. 2000. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge: Cambridge University Press.

Woolgar, Steve, ed. 1991. *Knowledge and Reflexivity: New Frontiers in the Sociology of Knowledge.* Vol. 20. London: SAGE Publications.