# Computer says 'no'

## On the materiality of software in organising tech work

*Paula Bialski*

**Paula Bialski** is Professor of Digital Sociology at the University of St Gallen, Switzerland.

**ABSTRACT**

**This article examines the materiality of software in shaping the organisation of software work, with a focus on how software's technical constraints – such as bugs or legacy code – impact daily workflows, decision-making processes and power dynamics. This article argues that software actively organises labour by influencing how developers negotiate their tasks and assert power in relation to management, colleagues and clients. Drawing on two ethnographic examples, this article highlights how the material properties of software mediate tech work practices and professional identities, contributing to a deeper understanding of the social and technical dimensions of contemporary software development.**

**KEY WORDS**

**software; legacy code; software bugs; ethnography; corporate work culture; science and technology studies**

DOI: 10.13169/workorgalaboglob.19.1.0067

## Introduction

Around the early 2000s, an English comedy series called *Little Britain* featured a recurring sketch with a receptionist named Carol Beer (played by David Walliams). Carol is depicted as being completely bored and useless, and she blames her lack of ability to do anything or solve anything on the computer she is working with. When a client comes into the office to ask for something, she gives a long-winded answer with the punchline 'computer says no'. Although Carol was a receptionist and not a software engineer, the phrase 'computer says no' has since been 'memeified' by software

workers[1] in various settings as a way of expressing the dynamics of corporate software development culture. For example, software developers have used the meme in instances when they have been unable to deliver or finish their software features on time because of the materiality of the software itself – when they have faced complicated, convoluted code, old and ageing 'legacy software', a range of unpredictable bugs or other software-specific factors. 'Computer says no' also became, for a while at least, the phrase that helped developers communicate their superior competence over others. For example, developers used 'computer says no' to indirectly communicate that they had the authority to tell others what the computer's limitations were, helping develop resistance to management methods, logics of excellence and perfection, and the commercial deadlines that drove the corporate software office.

Software is part of a world of relations (Kelty & Erickson, 2015). The many properties of software (which shift depending on its context, of course) concretely affect the world of people working on and with it. Software also affects how tech workers work, how they define themselves, and how they use these properties to relate to their work environment: to their management, customers and colleagues.

The goal of this article is to take the 'tech' in tech work seriously – taking apart the problems and forms of relations that software development affords its workers. In this article, I will explore how software, particularly its specific features and limitations, influences the daily work practices of software workers. I focus on how software shapes workflows, decision-making processes, and power dynamics within teams and between employees and management. Specifically, I examine two dimensions: first, how software workers navigate and negotiate the constraints imposed by software to accomplish their tasks; and second, how they leverage software's affordances and limitations to exert influence or resist managerial control.

In doing so, this article highlights the importance of looking at what software workers build and how – and how their software affects the nature of their work. Ultimately, I argue that an understanding of 'tech work and its discontents' should also look at the material resistance of software technologies themselves. By highlighting the specific, lived experiences of software developers, this paper offers a nuanced and grounded understanding of how the materiality of software shapes work practices, power dynamics and professional identities in contemporary tech environments.

In contributing to the discourse on tech work, this article also builds on insights from science and technology studies (STS), particularly regarding the materiality of software, which is often underemphasised in the sociology of labour. More recently, Bruni (2024) has noticed how 'work' and 'organisation' have progressively dropped out (at least in explicit terms) from the STS debate. By drawing on STS concepts of technological agency, this research highlights how the material properties of software – its bugs, legacy systems and inherent limitations – actively shape work practices, organisational dynamics and workers' agency. This approach not only enriches our

1 While I understand that 'tech work' can encompass a variety of jobs, in this article when I refer to 'tech workers' I mean software engineers working with, building or maintaining software code. From now on I will use the term 'software work' to avoid confusion with other types of tech work that might be featured in this Special Issue.

understanding of tech work but also bridges a gap between STS and labour sociology, offering a holistic perspective on the interplay between technology and labour in contemporary workplaces.

This article will begin by providing a brief theoretical background on how materiality has been studied in tech work. It will then draw on two ethnographic examples of the day-to-day experiences of those who work with and are shaped by the materiality of software. These ethnographic vignettes serve as a bridge between theory and practice, providing tangible examples that illustrate and expand upon the abstract concepts discussed by previous scholars.

## Technological determinism, affordances and actor networks

The concept of materiality and how it influences work practices is not something new, and it is a rich area of study, particularly in relation to research on craftsmanship, technology and organisational behaviour.

As Wanda Orlikowski highlighted in 2007, developing ways of dealing with materiality in organisational research is 'critical if we are to understand contemporary forms of organising that are increasingly constituted by multiple, emergent, shifting and interdependent technologies' (Orlikowski, 2007:1435). Indeed, materiality is entailed in every aspect of organising,[2] from

> *the visible forms – such as bodies, clothes, rooms, desks, chairs, tables, buildings, vehicles, phones, computers, books, documents, pens, and utensils – to the less visible flows – such as data and voice networks, water and sewage infrastructures, electricity, and air systems. (Orlikowski, 2007:1437)*

While there is a range of scholarship that looks at how materiality affects the relationship between workers or 'practitioners' and their objects, for example in craftwork (Sennett, 2008; Ingold, 2001), I wish to focus here on previous studies of engineers and software workers, looking at the way in which software itself shapes the way people work. One of the flagship examples of this type of study is Lucy Suchman's work (1987; 2007), which discusses the material aspects of human–computer interaction, particularly how physical and digital environments influence the practices of workers. Drawing on her research from the late 1970s to mid-1980s at Xerox Palo Alto Research Center (Xerox PARC), she introduced the concept of 'situated action', whereby the materiality of tools and environments affects the actions and decisions of workers. This concept enables us to understand actions (like the work of software engineers) as being influenced by the specific context and circumstances in which they occur (for example, an office, with particular codes for a specific hardware product that has a specific history).

2 In the context of this Special Issue, 'organising' could easily be misunderstood as referring to collective action by workers. This article understands 'organising' the way that organisation scholars do – as shaping, framing and giving agency to certain practices.

Dourish (2001) also explored how the physical and digital materiality of systems influences human interaction. He argued for a view of technology that considers the embodied nature of interactions, in which the material aspects of the digital world shape practices. While Dourish did not study software workers specifically, his emphasis on how context, embodiment, and social practices shape interaction with technology is useful here – showing that what software does to its users also does something to its creators. Fuller (2003:20) explained this relationship by proposing that software constructs 'sensoriums': that each piece of software 'constructs ways of seeing, knowing, and doing in the world that at once contain a model of that part of the world it ostensibly pertains to and that also shapes it every time it is used' (ibid.). The code that builds and maintains software has certain constraints. So-called 'legacy code' is one example of a constraint. Legacy code is software code that is inherited from an earlier version of a system or application. It is often outdated, written in older programming languages, or uses obsolete frameworks, and can be difficult to understand, maintain or modify. Legacy code may lack proper documentation and be tightly coupled with other parts of the system, making it prone to bugs and challenging to integrate with newer technologies. What is also key here is that legacy code holds a certain history: namely, past programmers' ways of designing, coding and envisioning solutions. After months or years, engineers leave their workplaces and projects behind, but their ideas and visions linger in the code they once wrote. These ways of coding also bequeath certain constraints onto the developers who take their place because legacy code cannot just be thrown away and is often crucial for an organisation's operations, as it continues to perform essential functions. This example shows that technology (legacy code) can literally put constraints on how developers work.

The influence of technology as an artefact, or a form of 'technological determinism,' was part of the 20th-century commentary on the rise of technology (the written word, typewriter, or television) (see Kittler, Von Mücke & Similon, 1987; McLuhan, 1964; Mumford, 1934, 2010). For technological determinists, technologies prefigure a range of social effects. For example, Kittler's post-humanist approach rejects the notion of the individual human actor or subject as a given. Instead, he examines how technologies enable, transform, and potentially erase both the language and embodiment of subjectivity (Gane, 2005).

This idea of prefiguration came with its critics: 'when "technologies" are assumed as a given or are lent the status of an independent variable, it is all too easy to inscribe specific effects or outcomes to their usage' (Russell, 2007:132). As Jenny L. Davis explained: 'Actor-network theory (ANT) arose in response to the technological determinism promulgated by McLuhan and his contemporaries' (Davis, 2020:72).[3] Here, ANT understands that technology is powerful but sees humans as similarly so.

3 Marshall McLuhan, with his claim that 'the medium is the message' did just that: showed us that we should look at the effect of the medium on people watching it. Transferring McLuhan's claims to the understanding of software, we could speculate that he would want us to understand the influence of software itself (code and various forms of legacy code, algorithms, libraries, etc.) rather than just consuming it as a software product (e.g. the GPS system in our cars).

Through this lens, humans and technologies co-constitute themselves, and interactions and networks of humans and technology are 'power-laden and political' (ibid.: 82).

Other significant contributions to the research on computing cultures include studies that extend beyond focusing solely on engineers. These works (mentioned below), attribute agency[4] to the material objects involved, revealing that 'values, opinions, and rhetoric are embedded in codes, electronic thresholds, and computer applications' (Bowker & Star, 2000). Such studies also emphasise that infrastructure and artefacts possess 'politics', influencing us and acting as integral participants in our social interactions (Latour, 1990; Winner, 1993). This perspective underscores the importance of ethnographic research that considers seriously the material aspects of what is worked on, such as software, code, data, servers and algorithms.

Additionally, sociologically oriented branches of software studies have enriched our understanding by demonstrating that software encompasses more than just a series of commands and the systemic relationships they produce. Instead, software represents a social process (Mackenzie, 2006). Software structures and makes possible much of the contemporary world (Fuller, 2008:1), hence the need to understand the 'stuff of software'. Using the example of open-source code, Fuller shows how it interrelates with the world of work: 'how class libraries function as a form of solidarity between programmers in minimising labour-time' (Fuller, 2003:24). Here, Fuller explains the importance of understanding software and how 'social' it is: 'We can only generate social software in its full sense through fundamental research into the machine, numerical, social, and other dynamics that software feeds in and out of' (2003:26). Other, more contemporary, ethnographic research on how code and computer hardware affect the work of engineers has followed. Johnston (2009), for example, looked at how antivirus experts are both 'authorised and empowered by the behaviour of computer code' (Johnston, 2009:9) and O'Donnell's (2014) research investigated, among other things, how the world of game developers is shaped by the race to optimise the processing power used in gaming (2014:112).

To analyse digital cultures and digital work effectively, we must recognise software as both a social and technical artefact, carefully considering the material intricacies involved in its creation and maintenance. This means that the study of software's 'affordances' should be a key focus. As Davis (2020) aptly points out, we should explore not just what artefacts afford but also how they afford. Moving beyond a binary model of affordance – which tends to view objects in terms of what they either enable or constrain – we should instead ask, 'How does this object afford?' By shifting our inquiry from 'what' to 'how,' we begin to understand affordances as fluid and evolving, rather than fixed and binary (Davis, 2020:85).

When applied directly to tech labour, technologies can control or shape the speed at which a worker works (Callaghan & Thompson, 2001; Cohn, 2016), the duration or intensity under which the developer works (Tan & Weigel, 2022) and the hierarchies

4 With "agency" here I mean social action as being distributed among humans and non-humans. This understanding can be found in numerous approaches from both actor network theory (ANT) to science and technology studies (STS) (see: Latour, 1990; Law, 1992).

that emerge in the workforce (O'Donnell, 2014; Amrute, 2016). While not wishing to paint a simplistic picture of technological determinism at the workplace in this article, I would like to offer a more nuanced understanding of the role of the materiality of software, including its affordances and resistance, on work practices.

In summary, the literature discussed in this section draws attention to the critical role that technology, with its material specificities, plays in shaping work practices. From the craftsmanship metaphors of Sennett to the embodied interactions highlighted by Dourish, and the socio-technical dynamics explored by Suchman and Fuller, these scholars collectively emphasise that technology is far from a neutral backdrop to work. Instead, it is a powerful agent that influences, constrains and enables the activities of tech workers. This body of work makes it clear that any analysis of tech labour must account for the material and social dimensions of the technologies involved. In the next section, I will build on this foundation by presenting empirical examples from the field, illustrating how these theoretical insights manifest in the everyday practices of software engineers and other tech professionals.

## A note on methodology

This article draws on a six-month situated research project at a Berlin-based corporate software company that I call 'MiddleTech' (Bialski, 2024),[5] specialising in mapping, routing, and navigation software. During this period, I engaged in both observation and participatory research among software developers and their managers. My focus was particularly centred on teams responsible for both front-end and back-end aspects of routing and navigation software development. Beyond the six months of situated research, I also conducted sporadic research over two years (2016–2018), which included research visits, lunches, online chats, and phone calls to MiddleTech. I also travelled to San Francisco to conduct discussions with software workers in Silicon Valley (July–September 2016). While I do not draw specifically on the latter, this comprehensive approach provided insights into the everyday work practices and cultural norms that characterise corporate tech environments, not only in Berlin but also, as my research extended to show, in Silicon Valley and other tech hubs.

MiddleTech belongs to what I term 'Medium Tech' companies (in contrast to 'Big Tech'). These large companies go widely unnoticed in the popular imagination. They are not small or medium enterprises (a category that most start-ups fall into at first) but large companies with over 1,000 employees. At the time of my research, MiddleTech housed about 1,000 employees in its office in Berlin with another 7,000 working around the globe. The company's objective was to make digital maps and to provide location data and other services to individuals (in the form of a navigation app on your phone or in your car) and other businesses (in the form of location data needed for building certain software). Located in over 20 cities, with its largest offices in Berlin and Chicago, the company had a 30-year history of growth, acquisitions, and rebranding. Having

5 The research in this article is also featured in the book *Middle Tech: Software Work and the Culture of 'Good Enough'* (Princeton: Princeton University Press, 2024).

existed since the mid-1980s, in late 2015, six months before I arrived, MiddleTech was sold to a consortium of German car manufacturers, which, at the time of writing, held around 75% of the company's shares. MiddleTech was, and still is, a company built on years of history, founded on a mix of programming styles, legal regulations, different work practices and methods of conflict negotiation all brought together under a single sleek, modern, community-garden-covered roof. The company was markedly different from powerful tech companies like Google or Microsoft, which have 'scaled up' in size to employ hundreds of thousands of people globally, as well as in 'width', offering a broad range of products from operating systems to devices to AI platforms, as well as AdTech and programming frameworks. This size seemed to matter: I observed that the number of employees in a company was often tied to differing levels of employee engagement, management style and senses of personal accountability.

In my observational methodology, I tried to mirror the daily work rhythms of developers. I would arrive at the office around the same time that others would arrive – 9.30 or 10 am at the latest. This allowed me to assess and then capture any event or ritual that might be happening that day. I situated myself within one specific team (for example, a team working on navigation software specifically for electric vehicles) and my research would rotate between teams every few months. The evidence presented in this article was derived from field notes and interviews.

## Two vignettes

In this article, I employ two vignettes to illustrate the complex ways in which software organises tech work and mediates power dynamics. The first vignette, featuring a back-end[6] programmer named Jelena,[7] highlights the significance of unit tests in structuring workflow and fostering collaboration, while the second vignette, detailing a management meeting at MiddleTech, underscores the tensions between managerial expectations and the practical constraints of software development, revealing how software workers navigate and resist these pressures through their technical expertise.

### Software as organiser

I will start with a vignette that highlights the role of unit tests in organising tech work. While this is only one example of the type of work a developer performs with software, it draws attention to how software practices shape the work environment and influence the interactions between software workers and their code.

It was summertime in 2017 and Jelena and I walked around the corner from MiddleTech to the Indian restaurant nearby. Jelena was an outspoken C++ programmer from Romania who worked on electric vehicle routing. During our lunch, she started explaining what unit tests are: tools that developers create to verify that their code

6 In contrast to front-end programming, which focuses on creating the visual and interactive elements of software that users interact with directly, back-end engineers build and manage the server-side logic, databases, and algorithms that help software run.

7 All names in this article have been changed to keep the confidentiality of my interlocutors.

works as intended, especially in a collaborative environment where code interacts with others' work. For instance, in the context of a calculator program, unit tests would ensure basic operations like addition or handling errors function correctly. These tests serve as safeguards, ensuring that a developer's code does not conflict with or break the work of others. As Jelena explained,

> *Since all our code interacts with code from other developers, it's crucial to protect it. Without protection, someone might write competing code that conflicts with yours. Your code is always competing with other code. Unit tests provide coverage. They are tools that ensure nothing goes wrong with what you've written … you come up with a list of unit tests that seem reasonable for that feature. For this calculator, I'd write tests to check if all the operations (+, –, etc.) function correctly. I'd also try to cover all the edge cases. For instance if someone makes a mistake and divides by '0', I'd make sure the operation doesn't crash.*

I questioned what Jelena meant by 'reasonable', and she explained that

> *There is no consensus for anything. People have different approaches. Different sets of rules, from company to company, from department to department. The company that I worked for previously found unit tests a waste of time. So you really have one set of rules, and then you are forced to abandon them. Rules are put together by the architects of a system, or the lead developers, and it's really a preference of opinion. And in our profession what we consider correct changes over time! And even if you get used to a system in your one company, these things can change. It's normal to have these things evolve.*

I highlighted to Jelena that the notion of code interacting with other developers' code is really important in how developers relate to each other. She replied,

> *Yes, we are constantly breaking each other's stuff. What you are creating communicates with other code that others are building … It's like the house metaphor. When building a house, the plumber works on something, and then the carpenter. Often this doesn't happen at the same time, because their work would conflict with one other. But we do it at the same time.*

This vignette shows how unit tests are not merely technical artefacts; they are pivotal in organising tech work. They afford a form of structure to the development process, ensuring that code functions correctly and does not disrupt the collaborative efforts of a team. This organising role of unit tests underscores the material aspects of software – it imposes constraints, necessitates specific practices, and requires continuous maintenance. The immaterial qualities of software also surface, particularly in how unit tests serve as a bridge for communication and negotiation among developers. We can thus see first that the material aspects of software shape what Jelena and her teammates do and how they interact, and second that software is a relational object that mediates interactions and defines the boundaries of acceptable practice within the tech work environment.

Jelena also mentioned that the rules and practices around unit tests can vary and the knowledge that developers need to acquire must adapt to the practices of their

company or team. This variability underscores the adaptable nature of software development practices and how they are tailored to fit the specific needs and constraints of different organisational contexts. It also points to the evolving nature of best practices in the tech industry, indicating a continuous learning and adaptation process among software workers.

Jelena's comments about code interactions and the necessity of constant code reviews also highlight the interdependent nature of software development – and how software instils a need for collaborative work practices. Having 'no consensus on anything' means that developers are expected to continually account for the impact of their code on others, which sometimes works and at other times leads to conflict. This interdependence can also be seen as a form of power, where the quality and stability of one developer's work can significantly affect the entire project's success. Additionally, the variability in how unit tests are valued across different companies illustrates how software can be used to exert control or, conversely, how it can empower workers. In environments where unit testing is emphasised, developers wield a certain degree of power – they control the quality and stability of the code, which in turn affects the entire project's success. On the other hand, in companies where unit tests are deemed a waste of time, management exerts power by prioritising cost and efficiency over the robustness of the code. This dynamic illustrates how software is an object of conflict, used to negotiate power between software workers and their managers. By 'power' I here mean the power to decide what is done and how, where both managers and software workers use software as an object to shape not only how work is done but also their own role in the broader organisational culture.

My discussion with Jelena about code review also touches on how software development practices contribute to the formation of professional identity among software workers. Jelena's reference to the constantly evolving nature of best practices and the absence of a stable consensus in software development reflects the immaterial aspect of software – it is in flux, shaped by the collective practices and decisions of those who work with it. In practice, this fluidity means that software workers must continually adapt, learning new methods and redefining their roles in response to the shifting landscape of software development. This constant adaptation and the associated learning processes are central to how software workers define themselves and their place within the industry.

This vignette illustrates how software, through practices like unit testing, acts as both a material and an immaterial force within tech work. It also sheds light on the power dynamics within organisations that influence how developers approach their work and the inherent challenges they face in balancing quality with efficiency. Ultimately, the vignette draws attention to the importance of understanding the material and social dimensions of software development, making it possible to fully grasp its impact on tech work.

This is not to say that software alone structures the work of engineers. This vignette also reveals that the approach to unit testing is heavily influenced by the organisational culture and priorities. For instance, due to its resource constraints, Jelena's previous company considered unit tests a waste of time, whereas MiddleTech values them highly. Management decisions and company policies can exert power over the methodologies

and practices adopted by developers, shaping their work processes and priorities. What I wish to highlight here is that, through understanding the constraints and affordances of their code, developers do hold a certain power over management. In the following section I will try to outline how codes (and especially software bugs) help structure the power relations in software work.

## Worker pushback

The second vignette was taken from meeting observations and a post-meeting discussion that took place during the second summer of my fieldwork (in 2017), when I was observing the 'routing and navigation' team at MiddleTech.

Every week, Simon, the manager of the entire routing and navigation team, would organise a number of weekly meetings, called 'sync' (synchronisation) meetings, with his middle managers: one of them was his 'PO' (product owner) sync meeting, which addressed their client-facing roles and aimed at keeping their developers 'aligned' with the customer's needs. The other sync meeting was with his 'team leads,' who were not product owners but rather the more technologically savvy group, who had a technical overview of what their developers were doing and often took on a programming role themselves if necessary.

Within the PO sync meeting, Simon would talk to the participants about ways they could reach higher targets to build more robust software – namely, to reduce the number of bugs and errors that their team was producing (which Simon would be able to see from various analytics that were regularly produced about their software by an analytics team).

Throughout both of these meetings, Simon would use one word quite frequently. This was 'alignment'. He wanted things to be 'aligned'. I got a feeling that the term alignment was used to communicate control – that when Simon asked others to align, he meant that they should put things into their correct positions relative to his goals of how a company should be run, of how many bugs should be eliminated. Synchronisation and alignment were somewhat synonymous. Sync meetings were also about aligning the developers' practices, operations, and activities to meet Simon's plans.

But things were not always in line and in sync with Simon's plans. The software developers and the software they were working on had other speeds and limitations. While this following vignette does not feature software developers themselves, I insert it here in order to highlight the tensions that those representing the software engineers face. Product owners are go-betweens, communicating the needs of the developers to their managers and customers, and vice versa.

About ten people gathered in a larger conference room on MiddleTech's fourth floor for a PO sync meeting. Simon, the manager, began the meeting, discussing the 'KPM' resolution. In a company like MiddleTech, efficiency is measured in something called a 'KPM', or 'key performance metric' (sometimes called the key performance indicator). The KPM is supposed to measure the level of output of production, and what a KPM actually 'is' differs from industry to industry. A KPM in software development might be the number of features completed on a certain day (the more, of course, the better), or the number of bugs found in a piece of software (the lower, of course, the better). According to my observations, a KPM was perceived as a tool with a

variety of purposes: it may be employed to make the intangible work of software development very tangible through quantification; it serves as a tool to help control the work of the middle managers, who then try to quantify the work of the developers they supervise; it is also a way of showing off to customers and investors (look at our KPMs!); and finally it is a tool for judging the productivity and efficiency of a team. In this case, Simon was applying the KPM metric to the number of bugs reduced.

In the PO meeting, Simon began trying to get the POs to pressure the software developers to fix more bugs, aiming to improve the KPMs. Simon emphasised the ongoing challenges with their new software product and sought the POs' help in motivating developers. He stressed the importance of everyone in the company thinking about how to 'add value' to the company which should include reducing the number of bugs in the software product.

Gavin, a product owner, expressed the view that developers often prefer to focus on coding and fixing issues, rather than considering broader company goals, stating, 'In my opinion, a developer would prefer to just fix stuff. To get into the code …'

Alessandro (another product owner) then said, 'I think we sometimes just fix it because the customer wants it', meaning that the customer pressures developers to 'keep fixing', even though it might not be necessary to fix anything for the overall functionality of the software. When a customer says 'keep fixing' in such a context, according to Alessandro, they are just creating work for their developers – work that they are paying for but is not in fact necessary. Simon then questioned whether fixing certain bugs would significantly improve the user experience, prompting Alessandro to recount instances where customers had insisted on fixes even when the POs believed them unnecessary.

This perspective drew attention to a perceived disconnect between what developers see as important (fixing immediate problems), the company's focus on 'adding value' and the customer's sense of entitlement.

After the meeting, I noticed Gavin and another product owner named Sophia debriefing in the hallway. When I approached them, they smiled and knew I had something to say about the meeting. I asked,

> *So what was that drama all about? I don't get what this conflict was about needing more autonomy to make decisions. Is that actually what you guys are fighting for?*
>
> *Gavin: I don't know. We just get fed up with talking to the customer over and over and over again and them just pushing back. When we tell them, 'this bug is not important', they just say 'well it is', so you have to do it'.*
>
> *Sophia: Do you know how many emails we get a week from the customer? It's always about finding ways to push back.*
>
> *Gavin: We start hiding bugs. There are ways to hide them. So sometimes it gets to be too much, and we start hiding them.*

I wondered what 'hiding' might actually entail.

> *Sophia: I get Simon's point: we can't just keep doing the same thing and expect different results. But I just don't know what else we can do.*

The discussion between Simon and the POs revolved around how software, particularly the bugs within it, drives the priorities and work processes of the team. Here, the manager (Simon), the POs and the developers seemed to have conflicting interests. While bugs are a shared concern among developers, product owners and customers, for each of these groups this 'concern' differs and takes on higher or lower priority at any given time. Here, a 'bug' can be conceived as a boundary object. Boundary objects, as defined by Star and Griesemer (1989), are objects that serve as a point of intersection or communication between different social worlds or groups. They are flexible enough to adapt to the needs and constraints of various groups but robust enough to maintain a common identity across contexts. As they explain, 'boundary objects are a sort of arrangement that allow different groups to work together without consensus' (Star & Greisemer, 1989:602). Boundary objects are crucial in collaborative working because they allow different groups to work together without needing a complete consensus – and in this case, developers, POs, and managers have varying priorities and perspectives on these bugs.

Each group interacts with these bugs in different ways: developers see them as technical challenges to fix; product owners see them as obstacles or priorities that need managing; and customers see them as problems that need resolution. Despite these different perspectives, bugs provide a common point of focus that all groups must address. Moreover, bugs have different meanings for different groups. For developers, a bug might represent a specific technical issue that requires a fix (and the 'fix' could be urgent or pushed off to another day or week, depending on the developer's understanding of the problem). For product owners and managers, it might represent a broader issue of resource allocation and customer satisfaction. For customers, it represents a failure in the product that needs to be corrected. This interpretative flexibility is a key characteristic of boundary objects, allowing them to serve different purposes for different groups.

This discussion between Simon and the POs was also significant for this article because it illustrates the way software engineers are perceived, as 'preferring to just fix stuff' and 'to get into the code'. What Gavin meant here was that software engineers do not want to think about 'adding value to their company' and prefer to organise their work around fixing what is not working, based on what they find immediately important from their perspective based on their experiences, needs, subject positions or role in the division of labour.

This vignette also shows how bugs – which unexpectedly arise from the complexity of interrelated and entangled code – help organise the work between different groups. The discussions between Simon, the POs, and the developers reveal how bugs mediate interactions, negotiations and power dynamics. Decisions about whether to fix a bug, how to prioritise it, and who has the authority to make these decisions are central to organising work processes.

## Power through the software object

Another important point in this discussion is that product owners like Alessandro constantly hear from their developers that something cannot be done, cannot be fixed, etc. In other words, they want to have the authority to say, 'this bug … can't be fixed'.

This means that they want to impose the limitations of what can and cannot be done on the customer's product, rather than have the customer decide what needs to be done.

This dynamic echoes the humorously frustrating scenario depicted in the Carol Beer skit from *Little Britain*, where the rigid response of 'computer says no' becomes a stand-in for inflexible, non-negotiable software limitations. Just as the Carol Beer character embodies the silly absurdity of blindly following what the computer dictates, Alessandro's wish to assert final authority over what is possible similarly highlights the tension between human decision-making and the constraints imposed by technology. In both cases, we can see that the decision-making process is influenced, and often constrained, by the capabilities – or perceived limitations – of the software, ultimately affecting how work is organised and how power dynamics play out between developers, product owners and customers.

This dynamic is not just about technical limitations – it is also a power struggle. Developers, armed with their intimate understanding of the code and the systems they work with, exert power over both managers and customers by controlling the narrative of what is and is not possible. It is a way of saying, 'I know the "computer says no"' and using that knowledge to assert their expertise and influence the decision-making process. Here, the developers' deeper understanding of the software becomes a tool of power, allowing them to shape outcomes according to their expertise, often challenging or even overriding the expectations of managers and customers.

Additionally, such moments undermine the manager, and at times the customers, due to their lack of knowledge of the software system in question. Software systems come with limitations, and developers both understand these limitations (resulting in being 'stuck' and not being able to fix a bug) or use these limitations in order to push back against management and customer pressure. It is also worth noting here that managers rarely have the same programming competencies as their engineers. Most managers I encountered were in the situation Simon was in: he knew how to program in an older programming language that was used back in the late 1990s but lacked the ability to sit down and build or fix a contemporary piece of software if a project was crashing. He generally knew what the architecture of his project should look like and what types of programmers were needed to build and maintain a project. A lot of his knowledge was passed down to him via his team leads, who were the more experienced software engineers between him and the regular programmers.

The need to control an uncontrollable profession produces a series of attempts to discipline the new profession, which remain only partially successful. Frederick Brooks explored this dynamic in *The Mythical Man-Month: Essays on Software Engineering* (1975/1995), which helped highlight the tensions between software developers and their management and customers. From the 1950s onward, new computing technology threatened the stability of the established managerial hierarchy. As more and more corporations began to integrate electronic computers into their data processing operations, it became increasingly clear that regular managers could not tackle a computer the same way they tackled other office innovations, such as complicated filing systems and tabulating machinery. Much like many other knowledge workers, computer specialists were quickly granted an unprecedented degree of independence and authority to step in and get the job done (Ensmenger, 2010:17). Situated research

around the software labour process built on Brooks's findings. For example, in Kunda's *Engineering Culture* (1992), managers are portrayed as attempting to gain back a sense of power by promoting a given 'work culture' among programmers, which includes eliciting, channelling, and directing the creativity of programmers under slogans encouraging them to innovate. A more contemporary example here can be drawn from different games that software development managers impose on their workers in order to help developers tackle problems at work such as software bugs (Wu, 2024).

The phenomenon of workers pushing back by slowing down, taking it easy, or ignoring their managers' directives is not limited to software work, of course. Perhaps one of the earliest descriptions of this type of protest was found in Taylor's notion of 'soldiering', or working more slowly than is prescribed by management (Taylor, 1919:30). While Taylor used this term to denote the 'greatest evil' that a worker can do, organisation scholars, particularly in anthropology and sociology, have studied such behaviour in the context of how decisions are made, how resistance and conflicts at work emerge, or how various forms of knowledge and power are employed and ignored in corporate cultures (see, for example, Burawoy, 1982; Courpasson et al, 2012; Beverungen, 2019). At MiddleTech, the developers had a variety of reasons not to fix the bugs that were assigned to them: at times they were stalled because of the resistance of software (it was not possible or too difficult at that given moment); or at other times, they used the excuse of the complexity of the software to 'soldier' and work more slowly, even adopting the practice of 'quietly resisting work by withdrawing from it' (Paulsen, 2014:2). In each case, the 'stuff' of software could be seen to be a powerful force in shaping how labour was organised.

## Conclusions

In examining the multifaceted role of software in organising tech work, this article underlines the interplay between technology, labour practices, and power dynamics within the corporate tech office. By focusing on the lived experiences of software workers, I hope to have provided a nuanced illustration of how software shapes, and is shaped by, the social and organisational contexts in which it operates.

By bringing you into two moments at MiddleTech, I have tried to highlight how software is not just a passive tool but actively influences how software workers define their roles, interact with management, and collaborate with colleagues.

The ethnographic vignette featuring Jelena highlights the centrality of unit tests in structuring the workflow. Unit tests serve as a crucial organisational tool, ensuring code reliability and fostering a culture of collaborative scrutiny. However, the practices surrounding unit tests vary widely, reflecting the adaptability required in different organisational contexts. Furthermore, the vignette reveals how software development is deeply interdependent. The interactions between different pieces of code necessitate constant coordination and negotiation among developers. This interdependence can be a source of conflict but also a form of collective power, where the quality and stability of one developer's work can significantly impact the entire project.

The tensions observed in the 'sync' meetings at MiddleTech illustrate the challenges of aligning individual and organisational goals with the limitations and affordances of

software. Managers like Simon use devices such as KPMs to control and quantify productivity, yet these measures often clash with the realities of software work. This made it possible to witness how the developers and their product owners pushed back against the unrealistic expectations of their clients and top managers. Additionally, the strategic use of software's limitations to resist managerial pressures highlights the dynamic power struggles within tech work.

In essence, this article argues that understanding tech work requires a serious consideration of the materiality of software (Mackenzie, 2006). The practices, principles and limitations of software development are not merely technical concerns; they are deeply embedded in the social fabric of the workplace. By taking the 'tech' in tech work seriously, we can better appreciate the intricate ways in which software organises labour, shapes power relations, and influences the collective consciousness of software workers.

In light of the ongoing scholarly discussions highlighted in this special issue and beyond, this paper also sheds light on the material dimensions of software as a critical factor in understanding tech work. There has been much focus on the figure of the software engineer and the broader tech workforce (e.g. Johnston, 2009; Amrute, 2016; Dorschel, 2022), with only a few researchers (e.g. Cohn, 2016; Mackenzie, 2006; 2017) exploring the material agency of the software they are building.

By ethnographically examining how software workers interact with and navigate the resistances posed by software – through bugs, legacy code and other complexities – this study adds a crucial dimension to the discussion of tech labour. It highlights that the materiality of software is not just a backdrop but a key player in the organisation of tech work, influencing how power is distributed, how work is performed and how workers perceive their roles within the industry.

Moreover, by focusing on these material resistances, this article provides insights into the ongoing conflicts and negotiations within tech workplaces, which are central to the current discourse on the proletarianisation of tech work (Steinhoff, 2022; Rothstein, 2022), the challenges of organising software workers (Bergvall-Kåreborn & Howcroft, 2013) and the potential for emancipatory movements within the industry. As software workers increasingly engage in struggles over workplace conditions, unionisation and broader social justice issues, understanding the material entanglements of their work becomes ever more critical. Circling back to the beginning of this article, the 'computer says no' anecdote reminds us that those with the knowledge to understand and navigate the complexities of software (namely, engineers) hold a significant degree of power. This power allows them to determine and communicate what the technology can and cannot do, often shaping the decisions and actions of others inside and outside the workplace, including managers and customers.

This article, therefore, invites further exploration into the socio-technical dimensions of software development, urging scholars to use ethnography to consider the material and social entanglements that characterise this field. Understanding these dimensions is crucial not only for addressing the discontents of tech work but also for envisioning more equitable and sustainable practices in the digital economy. By contributing to this conversation, this study hopes to bridge the gap between the technical and social analyses of tech labour, offering a comprehensive view of the challenges and opportunities faced by software workers today.

## ACKNOWLEDGEMENTS

I would like to thank Mace Ojala for your epic support, tips and editorial revisions to the final stages of this article as well as Norah Franklin and Ursula Huws for your copy-editing and proofreading help.

## REFERENCES

Amrute, S. (2016) *Encoding Race, Encoding Class: Indian IT Workers in Berlin*. Durham: Duke University Press.

Bergvall-Kåreborn, B. & D. Howcroft (2013) '"The future's bright, the future's mobile": a study of Apple and Google mobile application developers', *Work, Employment and Society*, 27 (6): 964–81.

Beverungen, A. (2019) 'Executive dashboard as a mediating technology of organization', in T. Beyes, R. Holt & C. Pias (eds) *The Oxford Handbook of Media, Technology, and Organization Studies*. Oxford: Oxford University Press.

Bialski, P. (2024) *Middle Tech: Software Work and the Culture of Good Enough*. Princeton: Princeton University Press.

Bowker, G.C. & S.L. Star (2000) *Sorting Things Out: Classification and its Consequences*. Cambridge, MA: MIT Press.

Brooks, Jr, F.P. (1975/1995) *The Mythical Man-Month: Essays on Software Engineering*. Boston: Addison-Wesley Longman Inc.

Bruni, A. (2024) 'Introduction: Work and organizing in scientific and technological phenomena', *Tecnoscienza–Italian Journal of Science & Technology Studies*, 15 (1):9–20.

Burawoy, M. (1982) *Manufacturing Consent: Changes in the Labor Process under Monopoly Capitalism*. Chicago: University of Chicago Press.

Callaghan, G. & P. Thompson (2001) 'Edwards revisited: technical control and call centres', *Economic and Industrial Democracy*, 22 (1):13–37.

Cohn, M.L. (2016) *Convivial Decay: Entangled Lifetimes in a Geriatric Infrastructure*. Paper presented at the *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, San Francisco, CA.

Courpasson, D., F. Dany & S. Clegg (2012) 'Resisters at work: generating productive resistance in the workplace', *Organization Science*, 23 (3):801–19.

Davis, J.L. (2020) *How Artefacts Afford: The Power and Politics of Everyday Things*. Cambridge, MA: MIT Press.

Dorschel, R. (2022) 'Reconsidering digital labour: bringing tech workers into the debate', *New Technology, Work and Employment*, 37 (2):288–307.

Dourish, P. (2001) *Where the Action Is: The Foundations of Embodied Interaction* (Vol. 210). Boston: MIT Press.

Kelty, C. & S. Erickson (2015) 'The durability of software', in I. Kaldrack & M. Leeker (eds) *There is No Software, There Are Only Services*. Lüneburg, Germany: Meson Press:39–56.

Ensmenger, N. (2010) *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, MA: MIT Press.

Fuller, M. (2003) *Behind the Blip: Software as Culture*. New York: Autonomedia.

Fuller, M. (2008) *Software Studies: A Lexicon*. Cambridge, MA: Mit Press.

Gane, N. (2005) 'Radical post-humanism: Friedrich Kittler and the primacy of technology', *Theory, Culture & Society*, 22 (3):25–41.
Ingold, T. (2001) 'Beyond art and technology: the anthropology of skill', in M. B. Schiffer, (ed.) *Anthropological perspectives on technology*. Albuquerque: University of New Mexico Press:17–31.
Johnston, J. R. (2009) *Technological Turf Wars: A Case Study of the Computer Antivirus Industry*. Philadelphia: Temple University Press.
Kunda, G. (1992) *Engineering Culture: Control and Commitment in a High-Tech Corporation*. Pennsylvania: Temple University Press.
Kittler, F., D. Von Mücke & P.L. Similon (1987) 'Gramophone, film, typewriter', *October*, 41:101–18.
Latour, B. (1990) 'Technology is society made durable', *The Sociological Review*, 38 (1):103–31.
Latour, B. (1993) *We Have Never Been Modern*. Cambridge, MA: Harvard University Press.
Law, J. (1992) 'Notes on the theory of the actor-network: Ordering, strategy, and heterogeneity', *Systems Practice*, 5:379–93.
Mackenzie, A. (2006) *Cutting Code: Software and Sociality* (Vol. 30). New York: Peter Lang.
Mackenzie, A. (2017) *Machine Learners: Archaeology of a Data Practice*. Boston: MIT Press.
McLuhan, M. (1964) *Understanding Media: The Extensions of Man*. Boston: MIT Press.
Mumford, L. (1934) *Technics and Civilization*. Chicago: University of Chicago Press.
O'Donnell, C. (2014) *Developer's Dilemma: The Secret World of Videogame Creators*. Cambridge, MA: MIT Press.
Orlikowski, W.J. (2007) 'Sociomaterial practices: exploring technology at work', *Organization Studies*, 28 (9):1435–48.
Paulsen, R. (2014) *Empty Labor: Idleness and Workplace Resistance*. Cambridge: Cambridge University Press.
Rothstein, S.A. (2022) *Recoding Power: Tactics for Mobilizing Tech Workers*. Oxford: Oxford University Press.
Russell, B. (2007) '"You gotta lie to it": software applications and the management of technological change in a call centre', *New Technology, Work and Employment*, 22 (2):132–145.
Sennett, R. (2008) *The Craftsman*. New Haven: Yale University Press.
Star, S.L. & J.R. Griesemer (1989) 'Institutional ecology, translations' and boundary objects: amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907–39', *Social Studies of Science*, 19 (3):387–420.
Steinhoff, J. (2022) 'The proletarianization of data science', in M. Graham & F. Ferrari (eds) *Digital Work in the Planetary Market*. Boston: MIT Press:191–208.
Suchman, L. A. (1987) *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge: Cambridge University Press.
Suchman, L. (2007) *Human-Machine Reconfigurations: Plans and Situated Actions*. Cambridge: Cambridge University Press.
Tan, J. & M. Weigel (2022) 'Organizing in (and against) a new cold war: the case of 996.ICU', in M. Graham & F. Ferrari (eds) *Digital Work in the Planetary Market*. Boston: MIT Press.
Taylor, F.W. (1919) *The Principles of Scientific Management*. New York: Harper & Brothers.
Winner, L. (1993) 'Upon opening the black box and finding it empty: social constructivism and the philosophy of technology', *Science, Technology, & Human Values*, 18 (3):362–78.
Wu, T. (2024) *Play to Submission: Gaming Capitalism in a Tech Firm*. Pennsylvania: Temple University Press.