CONTENTS

Introducing Good Enoughness

The notion of "good enough" is strange: often it means that we have given up on the desire to be great, or even excellent, and sorrowfully succumb to compromise. Even though the phrase "good enough" means that there is "enough goodness," and that things are generally fine, the phrase also evokes failure or giving up and embracing mediocrity. I bet you would quickly return this book to wherever it came from if on the back cover a reviewer wrote, "This book is not bad, not excellent, but just good enough." Or what if I told you that the software running in your car was good enough? Wouldn't that be slightly scary? Or what if a colleague or boss said that the job you were doing was good enough? "Good enoughness," a term I use throughout this book,[1] might have a pejorative ring to it. It connotes mediocrity, a failure to achieve more; it's something that we humans have learned *not* to desire. Yet this book offers another perspective on what "good enough" means by focusing on the regular, ordinary work of corporate software developers making regular, ordinary software, and on the complex decisions, everyday practices, hidden ethics, and implicit and explicit collective negotiations that make good-enough software possible. My point throughout this book is that achieving good enoughness is an incredibly complex and interesting endeavor.

1. I toyed with using the neologisms "good enoughing" or "good enoughness," yet chose the latter to stay consistent. Both are a bit awkward, but it was important for me to create a term that highlighted an unfolding and negotiated process. Throughout my book, "good enough" is less objective criteria, more a state, and for sure a practice. Both "good enoughing" and "good enoughness" could have worked.

The first moment I remember encountering good enoughness in my field was on a Friday afternoon during one of my first weeks of fieldwork at a company I call MiddleTech, a mapping and navigation software company in Berlin. It was getting close to 4 p.m., and happy hour was approaching. A few software developers were planning to meet up for beers across the street, and Marek (a front-end developer working on the Android navigation app) had not yet finished his code review. Much like any peer review, software developers have to review each other's code before submitting it to the main code base. It was getting late, and the other developers called to Marek: "Are you joining? Just give a +2 and come on!" They started laughing. Giving a +2 during code review meant giving the code a green light and integrating it into the working software system. A web developer on Marek's team later confessed that when he feels like leaving work and running off for a beer, he quickly goes through the code review system and just adds +2, +2, +2 to all the tickets waiting to be reviewed. Marek followed suit, and fifteen minutes later we were all sitting and sipping craft beer, enjoying the warm autumn Berlin weather.

The gesture of giving fellow developers a +2 in order to leave work was not done out of sloppiness, laziness, resistance, or protest, or at least not mainly so. Engineers care about the software they work on, and Marek was no exception. Marek was also not prone to political resistance against the demands of his labor process. Marek clicked on +2 that Friday afternoon because he knew his colleague's code was good enough. By clicking +2, he expressed an understanding that the code was good enough for now. Moreover, he knew that if anything went wrong, he would have the ability to come back and fix it later. Knowing when to stop and say something was good enough was not about not caring but about understanding the balance between care and compromise.

As my first encounter with good enough software culture unfolded before my eyes, it seemed counterintuitive, shattering my own stereotypes about what software production looked like. Weren't software developers supposed to be aiming for seamlessness and efficiency? It stood in stark contrast to the narratives I encountered earlier that summer, interviewing various technologists from the San Francisco Bay area—people at Facebook, the Wikimedia Organization, Mozilla, the Electronic Frontier Foundation, and a slew of entrepreneurs.

The Silicon Valley techies I encountered seemed to believe that technology had to be great, and that work on technology had to be hard and sweaty. I spoke with Eric, an older investor and entrepreneur in San Francisco whose

long career was based on liaising between venture capitalists and programmers. During my discussion with him in San Francisco, he explained, "Coders do it just for their *art*. They want to sit and perfect their little babies. Coders sit over their laptops and want to develop until it's done. The harder the project, the better. If they code something that's outta this world, they will get recognized for it. And it's that recognition they're after. Like, 'Hey man, you did it, you're the shit.'"

While Eric might have been an extreme stereotype of somebody with Silicon Valley tech fever, many engineers I met that summer in Silicon Valley fit his description: They were driven by a similar narrative to change something in the world with technology, to do something difficult, and to strive for a sort of aesthetic excellence. What I found striking was the repetitive narrative that software developers were dedicated to working into the late hours perfecting something "outta this world." Software was not just patched together to run, occasionally break down, and be maintained; it was meant to run, disrupt, and innovate all in one go. Within this cloud of Silicon Valley hype, I never could have imagined that a software developer somewhere, on a Friday afternoon, would give another software developer a +2 in order to go out for beers with their friends.

My long-term fieldwork at MiddleTech helped me understand that the discourse and practice of making excellent software under a hyped work ethic are at odds with regular, run-of-the-mill corporate tech offices, where software and software work practices are about being good enough rather than excellent. The corporate tech office—both in Berlin and, as I will discuss, in Silicon Valley and beyond—propagates and maintains a state of good enoughness, despite discourses stating the contrary.

I spent an intensive six months (with additional field visits and interviews spanning two years) observing and at times participating in the work of software developers at a Berlin-based corporate software company that makes mapping, routing, and navigation software. This research focused on the software developers and their managers in both the front-end and back-end routing and navigation teams. During my fieldwork there, I worked among hundreds of people.[2] On a daily basis, I would discover new people, new conversations, new departments, and new projects, all of which would send me down another interesting research path. I recorded these stories in

2. In this book you'll notice that I often describe the field by directly quoting various interlocutors. It is worth noting that the conversations I reference from MiddleTech were not audio recorded but taken from my field notes in which I paraphrased the discussions with my interlocutors.

my field diaries, both on paper and digitally, during my fieldwork and after I left the office. The latter helped me blend in with the people I sat next to: while hunched over typing away on my laptop, I was at times mistaken for a new programmer on the team. I concluded that at MiddleTech, software is an ephemeral object that needs to be only good enough to function until the next update. The people working on it are well aware of this fact and often don't feel too pressured to perform perfectly during the first, second, or even third iterations. As a consequence, software can never be great but is instead just, well, good enough.

Drawn directly from my observations in the field, this book joins recent efforts to complicate the discourse that software is seamless and awesome (and not just good enough), and that the corporate software worker needs to be driven to achieve excellence. As we have witnessed throughout the past, technology breaks: staff cutbacks cause media platforms to break,[3] in-car GPS systems cause catastrophic incidents (Lin et al. 2017), and chatbots "tell lies and act weird."[4] The stories we hear in popular media shape our understanding of digital technology as either a technosolutionist savior, a mediocre disaster, or a robot-apocalyptic nightmare. As many ethnographies hope to do, this book provides a more complicated, less sensationalist, empirical story of why software can't be perfect. My time at MiddleTech helped me highlight how the ethics of practice prevalent in corporate software cultures encourages a state of being good enough, where something (like software) or someone (like a software developer) needs to be only sufficiently competent to operate. As I will show throughout this book, good enoughness is an inevitable part of software culture that contrasts with the popular understandings of how software is built and what software is. Defining good enough is collectively negotiated in resistance to managerial ideology while fluctuating between care and compromise for what, with, and for whom one is building software. It is an aspect of German software culture but is also present in larger, aging corporate software companies globally, and it might be inherent in all software development.

3. Ryan Mac, Mike Isaac, and Kate Conger, "'Sometimes Things Break': Twitter Outages Are on the Rise," *New York Times*, Feb. 28, 2023, https://www.nytimes.com/2023/02/28/technology/twitter-outages-elon-musk.html.

4. Cade Metz, "Why Do A.I. Chatbots Tell Lies and Act Weird? Look in the Mirror," *New York Times*, Feb. 26, 2023, https://www.nytimes.com/2023/02/26/technology/ai-chatbot-information-truth.html.

Studying Software Developers

Before I dive into this book's central argument, I'd like to explain the origin of the thinking behind my book. My exploration of the culture of good enoughness first began as a quest to understand the fluctuating relationship between the production of technology and society. My research started by asking how "the society we live in affects the kind of technology we produce" (MacKenzie and Wajcman 1985, 2) and turned to the producers, designers, and programmers of technology and those who managed them. Focusing on the producers of technology, rather than the users, was not as self-evident as it might seem. Following a tradition of science and technology studies scholars, I ethnographically focused on an overlooked group of engineers rather than on the simplistic narrative of the lone-wolf innovator (Haigh and Priestley 2015).

MiddleTech was always meant to be an ethnography about how a collective group of people collaborate, communicate, care, and compromise in order to make software work. By getting to know their work hierarchies, their forms of interaction, and the micropolitics of their profession, I encountered the programmers' social world. As I will illustrate throughout the next chapters, good-enough software is achieved through collective software practices, where programmers learn the process of programming something in a good-enough way, which is part of their sense of belonging and engagement in their sociotechnical worlds. Negotiating what is good enough or not—through discussions, jokes, fights, and other practices—is an important part of the collective practice of corporate programming.

My research resonated with maintenance and repair research, which focuses on the programmer and those conducting the maintenance and repair. As Lee Vinsel and Andrew L. Russell (2018) reminded us, life with technology is usually far removed from the cutting edges of invention and innovation and is instead devoted to keeping things the same. Drawing on these researchers and their tropes, *MiddleTech* starts with an interest in the programmer: interest in the human condition of being engaged with the craft of programming, their relationship to their machine, and the way their work and their profession are negotiated within their community.

MiddleTech also became an empirical description of the material constraints of software work, where software cannot be perfect in practice due to certain forms of complexity in software production. Throughout the following chapters, I describe how old code, software's constant cycle of being updated, its architecture, and how it is designed and by whom all contribute

to the material complexity of software. As Marisa Leavitt Cohn (2016) has highlighted, our software and our software companies are aging. As our software ages, our software projects become more and more complex, evolving into multilayered beasts, "polluted" by programs, reports, files, or data that lose their purpose over time (Visaggio 2001). Much of the software we use today is built on years and years of effort by software developers who have managed to patch together a project to make it work. As our societies continue to strive for smarter systems (Halpern and Mitchell 2023) and better solutions to our problems, it is crucial to understand the faults in the technologies we so trust. Software's increasing complexity and age also challenge the relations between programmers, managers and their programmers, programmers and their code, and various other actors involved in the entire process. The moments when these actors have to negotiate care and compromise are also a crucial part of the story of our technological societies, and understanding this can help us as users, customers, and creators grasp the tricky materiality of software: that the tools we use are sometimes based on forgotten updates, lost pieces of code, and scrapped software projects, which, among other issues and mishaps, contribute to merely good-enough software.

Lastly, this book looks at the environment in which these material software practices unfold. In particular, I became interested in how corporate culture is shaped and reinvented (Kunda 1992) in the tech sector, both top-down through managerial discourse and bottom-up via the practices of engineers. My analysis zooms out to the corporate, organizational level, where understanding the power dynamics, work processes, and management dynamics within a corporate setting becomes central to understanding the culture of good enoughness—both how it is counterintuitive to various corporate narratives and rituals, and how it becomes negotiated on a day-to-day basis. We will witness the contrasting and chaotic priorities and understandings between designers, managers, and programmers working on the same product, which has been also observed in other ethnographies of software cultures.

While these other ethnographies look at how race and class are negotiated in corporate software settings (Amrute 2016) and how programmer work is organized (O'Donnell 2014), this book's specificity lies in its ethnographic account of the work cultures within older, aging companies. In the past decade, increasingly digitized Western societies have had an abundant need for programming work. Additionally, as tech companies grow bigger and become more established and embedded within our society, they are

here to stay—meaning they are growing older, adding a level of complexity to the code being worked on and produced. Taking into account that software is an "object subject to continuous change and lived with over time as it evolves" (Leavitt Cohn 2019, 423), one that does not sit still "long enough to be easily assigned to conventional explanatory categories" (Mackenzie 2006, 18), *MiddleTech* zooms in on a work culture within a growing and aging software industry and aims to give a more nuanced understanding of digital media as inherently made up of these mishaps and compromises, bugs and breakdowns, and wonky, half-baked, good-enough work and good-enough software. Thus, to understand good-enough culture, understanding the material agency of software is important, specifically in relation to how corporate software is still produced, repaired, and maintained.

Not Bad, Not Excellent

The notion of "good enough" in this book contradicts and complicates the discourses and normative orders of excellence and improvement that permeate the tech world and shows that there is a distinction between discourse (which includes metrics and management methods) and the everyday practices of software developers. Throughout the following chapters, we will witness how workers reject notions of excellence *in practice*, but I'd like to highlight that a hegemonic excellence discourse does exist *in theory.* Corporate software companies, like many corporate environments, propagate an ideology of excellence and improvement, both in relation to the software product they are building and regarding the type of work that goes into building a software product. But where do these normative discourses of excellence originate?

One of the best places to search for the roots of the narratives of excellence, perfection, and 100 percent–ness is management literature. Written for managers, usually by more successful managers or management scholars, these books and journals show what types of narratives permeate corporate culture. At MiddleTech, it was quite common to find this sort of management literature lying on a desk or tucked away on a bookshelf in the company library. For example, the *Harvard Business Review*, a key publication for managers and management scholars, is full of case studies in which clear "performance expectations" are set by managers and team members, "performance measures" are delineated by said managers, and finally, the goal of achieving "performance excellence" is (hopefully) met by the given team. The *Harvard Business Review* and other similar industry journals are

full of tips on how to foster or scale up a "culture of excellence" in the fastest way possible.[5] This type of rhetoric can also be found throughout management handbooks, one of the most prominent being Thomas J. Peters and Robert H. Waterman's *In Search of Excellence: Lessons from America's Best Run-Companies*, which despite having been written in the early 1980s, is still used today to help managers achieve "productivity through people" in order to become a "learning organization" (1982, 111) that experiments with and tries new things while striving to be the best.

More recently, Robert Sutton and Hayagreeva Rao (Stanford professors of Management Science and Organizational Behavior and Human Resources, respectively) promised to show managers "what it takes to build and uncover pockets of exemplary performance, spread those splendid deeds, and as an organization grows bigger and older—rather than slipping toward mediocrity or worse—recharge it with better ways of doing the work at hand" (2014, 20). In their book *Scaling Up Excellence: Getting to More Without Settling for Less*, "driving towards mediocrity" is seen as the first step to downfall, and Sutton and Rao are here to help companies foster a "relentless restlessness" that helps them constantly innovate (20).

As Paul du Gay explained, "Excellence in management theory is an attempt to redefine and reconstruct the economic and cultural terrain, and to win social subjects to a new conception of themselves—to 'turn them into winners,' 'champions,' and 'everyday heroes'" (1991, 53–54). This is done through a new form of management that emphasizes good corporate culture that can foster these "winners" and "heroes." Corporate culturalism, in its central argument, strives for an expanded practical autonomy of the worker. Yet as Hugh Willmott has pointed out, it aspires to "extend management control by colonizing the affective domain. It does this by promoting employee commitment to a monolithic structure of feeling and thought, a development that is seen to be incipiently totalitarian" (1993, 517). As I will show in the following chapters, engaging in good enoughness can thus be the software workers' way of regaining power over their "affective domain,"

5. See, for example, Tony Gambill, "A Leader's Challenge: Developing Teams That Have Strong Relationships and Excellent Results," *Forbes*, Sept. 14, 2022, https://www.forbes.com/sites/tonygambill/2022/09/14/a-leaders-challenge-developing-teams-that-have-strong-relationships-and-excellent-results/?sh=37d953766bb5, or Jeanine Murphy and Michael Sioufas, "How Agile Teams Can Pursue Technical Excellence," *McKinsey Quarterly*, Feb. 2, 2022, https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/tech-forward/how-agile-teams-can-pursue-technical-excellence.

rejecting the notion of excellence and settling for a software product and a way of working that's just good enough.

In my specific field at MiddleTech, I first noticed that when building critical software like routing and navigation infrastructure, corporate software developers work under the orders of managers who strive to build software that meets particular requirements and safety standards in order to gain certain levels of certification. These standards and certifications help order the world of software developers, their manager, and their customer (Bowker and Star 2000): it communicates to customers that the product (in this case software) they are using is seamless. At MiddleTech, software product managers gained certification from the International Organization for Standardization (ISO), a nongovernmental standards board that sets out various types of standards certificates for corporate software companies, including "quality management standards" and "IT security standards" among many others. In order to gain these certifications, products had to meet certain safety criteria or achieve certain metrics. Managers would meet these metrics by incorporating discourses and methods of working that would strive for perfection, particularly during the months leading up to a certification audit. Thus, to achieve seamlessness or these "great metrics," the office had to have a work discourse of excellence. In practice, developers negotiate what is good-enough work in order to meet these standards and metrics (or get away with not meeting them), but excellence is something managers still push as the overarching narrative to legitimize their own position and the ways of working around the office.

An Ideology of Improvement

Beyond the notion of excellence, another normative discourse that circulates around the corporate software office is the concept of improvement. If we accept that the update is a defining characteristic of software work culture, then we can also imagine that the notion of continuous improvement is essential to how programmers work. Each update carries the implication that developers can and should continuously iterate and improve on their product. That said, the ideology of improvement can be found everywhere in software work, materialized in the tools and methods that managers use to make software teams work better together and individual programmers code better. With hundreds of moving parts and dozens of teams of software developers carrying out work that their managers often do not understand, corporate software development processes have fostered cultures, rituals,

and forms of organization that get a product delivered, create accountability, and stabilize continuous improvement. One particular method is called "Agile", one iteration of which is called "Scrum," where software developers are meant to work in "sprints," two-week stretches devoted to particular tasks, which are broken down on Post-it notes on a whiteboard. In this method, the head of the development team reports on progress using software that includes a dashboard indicating the state of every project. "The manager could also show a graph of the team's 'velocity,' the rate at which the developers finished their tasks, complete with historical comparisons and projections" (Posner 2022). Developers also engage in a daily ritual called the stand-up, where they all stand around in a circle and take turns explaining how their work is progressing or how they are improving on each task.

This methodology emphasizes a culture of improvement, where discussions in team meetings, company meetings, and one-on-one manager-to-programmer and programmer-to-programmer meetings are often focused on how to improve something: how to improve a work process, how to improve communication, how to improve a piece of software, or how to optimize (improve!) an algorithm. The notion of improvement is woven through everything.

Additionally, in a company like MiddleTech, the velocity of improvement is quantified and measured using something called a KPI or key performance indicator. This performance indicator is not specific to software companies in particular (those who have worked in any other corporate environment have probably come across the term). As the metric is quite broad, a KPI has to be defined within each industry, based on something that a management team can track. In the past decades of software production, managers have attempted to track certain practices of the software developer's work, such as the number of lines of code a developer committed or entered into the system, or the number of features completed on a certain day (the more, of course, the better). Managers have also turned to software itself to measure KPIs by looking at the number of bugs in a software system or the code simplicity, meaning the number of independent paths code must take to run a piece of software (the fewer the better).

Progress is thus characterized by a distinct normativity of numbers (Anders 2015), meaning the use of numbers as norms for measuring a company's progress in fixing bugs, implementing innovative solutions, and introducing systems like the KPI or various company software tools to collect and process numbers in a standardized fashion. Numbers like KPIs are

essentially about projecting power and coordinating activity (Porter 1995, 44). In bureaucratic business corporations like MiddleTech, "quantification is simultaneously a means of planning and of prediction" (43), and there is great pressure for workers and their managers to conform to ever-increasing demands for "greater workplace productivity and enhanced efficiency modulated by computational systems that manage KPIs" (Rossiter 2016, 18). In other words, developers are being increasingly pushed into productivity by software-driven metrics, where KPIs and the real-time measurement of labor imply a constant acceleration described in terms of improved productivity. More specifically, the belief in the neutrality of certain metrics and measurements helps to enforce the corporate ideology that the software team and the software product can continuously improve and actually achieve excellence.

Excellence and Improvement and Reality

I discovered throughout my fieldwork that while these metrics, methods, and modes of excellence and improvement are present in the MiddleTech office culture, the reality is different. On a discursive level, corporate software environments can be understood as factories of so-called technological acceleration (Wajcman 2014, 16), where technology is constantly updated to improve and strive for excellence. Yet in the everyday, often mundane reality, software developers are more informed by good-enough principles and practices.

Good enoughness implies settling for the here and now, as opposed to accelerating forward to achieve something better. While in theory, an old software version is always being updated and improved, a software developer's practical tasks at the workplace don't necessarily have to be oriented toward improvement or some form of innovation. For example, a piece of navigation software that is shipped today might be full of bugs that slow down users. But the good-enough developer's tasks are often self-defined. One update might fix just two bugs instead of the imagined fifty. While cleaning up these few bugs might give users a more seamless experience, it can also cause other bugs to appear and other slowdowns to occur. Thus, while on a discursive level, managers and software workers may speak of accelerated improvement and innovation, in practice their relationship to this innovation and constant improvement can be quite ambivalent. Improvement doesn't always mean peak innovation and can instead be just good enough. This example also shows us that what is good-enough work is also a matter

of subjective estimation, normally arrived at by the developers who hold a more intimate knowledge of the code than their managers or the customers they work for.

The normative orders of excellence and the ideology of continuous improvement are strong forces driving the software industry and its socio-technical culture. This company ideology is something that is reproduced in day-to-day, face-to-face discussions, in meetings, conferences, and coffee breaks (Wittel 1997). Yet these ideologies are not necessarily something that everyone in the corporate software office believes in (Wittel 1997). While excellence and continuous improvement may permeate the office discourse, I observed that often neither software workers nor their managers really believe in the importance of excellence nor in the ability to continuously improve. For a particular ideology to survive, it is not essential that people actively support or believe in it. As Renata Salecl stated, "the crucial thing is that people do not express their disbelief. For them to abide by the majority opinion, all that matters is that they believe it to be true that most of the people around them believe. Ideologies thus thrive on 'belief in the belief of others'" (Salecl 2011, 10). What she means here is that people often do not believe in something but pretend to in order to avoid offending those who might believe in it.

Something similar in our context of software development is described in Frederick Brooks's *The Mythical Man-Month*. In his seminal text on software production methodology, Brooks (1975) explained that software development teams, particularly their managers, repeatedly plan for software projects to go well and be finished on schedule, when in reality projects are full of bugs and are always delayed. Brooks says that programmers hold beliefs or assumptions that "all will go well" or "that each task will take only as long as it 'ought' to take" (14), while in reality they often settle for good enough. As you will see in this book, when you candidly ask a manager or a developer if they really believe that a project will be finished on time, or if a piece of software will work seamlessly, they will emphatically say "no."

At MiddleTech, most developers and managers would openly (in meetings or job interviews, for example) express their belief in excellence, technological innovation, or the efficiency of production, while in reality, they practiced the opposite, meaning the work ethic and software ethic of good enough. Good enoughness, therefore, becomes an emergent cultural practice that happens in practice, juxtaposed to its more dominant other. These "others," which will reappear throughout this book, are excellence, technological innovation, and the efficiency of production.

Good Enoughness

The concept of good-enough software production is not one I coined myself but rather found in the field during conversations among developers at MiddleTech, in online hacker forums, or in software engineering literature. In their article "How Good Is Enough: An Ethical Analysis of Software Construction and Use," W. Robert Collins and his coauthors suggest that the software industry should "encourage reasonable expectations about software capabilities and limitations" (1994, 89), both among users and producers of software. This call to be "reasonable," as Collins and his colleagues explain, is about understanding "how good is good enough," a responsibility of the software provider or the programmers and their team. The term "good-enough software" highlights that perfect software for a complex system cannot be guaranteed in practice (Collins et al. 1994); thus, releasing software to the public will always be done under a good-enough principle, and will include some level of failure (Pelizza and Hoppe 2018). Good-enough software is, as Collins and colleagues explain, a principle that understands that every piece of new software can be assumed to contain errors, even after thousands or millions of executions.

In the mid-1990s, the concept of good-enough software was "getting a lot of attention" (Yourdon 1995, 78) in order to counteract the "we'll deliver high-quality, bug-free software on time" battle cry (78) that was sweeping the industry. In his short article in *IEEE Software* magazine, Yourdon explained that software engineers were shifting from working on proprietary, one-of-a-kind systems, developed according to schedules measured in years and funded by budgets measured in millions to software as a cheap commodity that can be made and reproduced relatively quickly. In other words, instead of making software for a shrink-wrapped CD to slip into our PC, the dawn of the internet brought programmers cloud computing and the ability to iteratively change the software in our fridges, phones, and desktops. Instead of perfecting and preserving a piece of software for eternity, the update became like a lifeboat or an eraser, enabling developers to fix their work at any time. In essence, the update gave the software developer the ability to settle for something good enough for now, only to be fixed later, which, as Yourdon explained, began "to challenge some of our basic assumptions about software development" (78).

Aside from software development, the good-enough principle has been used in psychoanalysis, pediatrics, urban studies, design, philosophy, biology, economics, and more popular self-help books. For example, using the

concept of the "good enough mother," the British psychoanalyst Donald Winnicott describes the caregiver who settles for "good enough parenting": recognizing the fragility of a baby but failing at meeting all of the infant's demands and one's own standards of the perfect mother. Through this failure, mothers allow their babies to find their own way of doing things (see Winnicott 1987 or Doane and Hodges 1992). The concept has also been taken up in medicine (Ratnapalan and Batty 2009), where practitioners argue that excellence in medicine can be achieved by ensuring results that are good enough rather than by aiming for perfection, or in psychological research methods, where researchers set standards that indicate what kinds of experimental outcomes are good enough (Serlin and Lapsley 1985).

In economics and organization theory, Herbert Simon coined the term "satisficing" to describe the decision-making process whereby individuals or organizations seek a satisfactory solution rather than an optimal one. Similar to good enough, satisficing is when people choose the first option that meets their minimum criteria for acceptability, rather than continuing to search for the best possible option. Simon argued that satisficing is a practical and efficient approach to decision-making as it allows individuals and organizations to conserve resources and make decisions quickly. He contrasted this approach with the idea of optimizing, which maximizes the benefits of a decision but can be time-consuming and requires extensive information and analysis: "Evidently, organisms adapt well enough to 'satisfice'; they do not, in general, 'optimize'" (Simon 1956, 136).

This approach also resonates with wider discussions around the prevalence of good enough in both biology and culture, where the evolution of many species on Earth was not optimal as Darwin believed, but they survived anyway in a good-enough state (Milo 2019). Other scholars called for society to embrace the "good-enough life" as a state that understands what "goodness" and "enoughness" mean (Alpert 2022). Alpert in particular links good enough to the human need to change our relationship with nature and ecology. He calls for a reduction in our production and consumption in order to live more in harmony with nature, building our "good-enough life within these good-enough conditions" (5). This plea for restraint and reduction goes hand-in-hand with notions around the "good enough job" (Stolzoff 2023), or the "smart enough city" (Green 2020), where "enough" means rolling back our need for acceleration and overproduction in our optimization-centric jobs or urban planning endeavors and "limiting growth" (Meadows et al. 1972). Here, being good enough can also be connoted with mediocrity, which, as Groth (2019a, 2019b, 2020a, 2020 b) highlighted, is increasingly

becoming a positive point of reference in different fields of practice. Keeping up with the midfield, earning a middle-range income, or being part of the middle class are powerful models for socioeconomic behavior and lifeworld interpretations (Groth 2019a).

Two Good Enoughs

As we can see, the notion of good enough has been used in various fields, including in organization studies and computer science (where this book is situated more closely). Rather than merely demonstrating that good enoughness exists, what I hope to highlight throughout these next chapters are the cultural aspects of good enoughness in practice. Over the course of my ethnographic observations, I noticed that two specific kinds of "good enoughs" emerged from my field, somewhat related but different at the same time. The first type of good enoughness addressed in this book relates to software itself. Software is a material product destined to be just good enough. Contrary to the seamless save-the-world technology promised in YouTube clips from product demos touted by CEOs like Elon Musk, Steve Jobs, or Mark Zuckerberg, software isn't all that it's cut out to be. When we look into software's constitution and how it's built and maintained, we see that at its core, it will always be merely good enough. Software is complex and made up of hundreds of lines of code that are constantly changing, constantly in flux. Due to this complexity, the people who work on software can never understand it in its entirety, which also makes these projects hard to manage, and as Brooks (1995) explained, they are hard to estimate in terms of scope and duration of completion. As I will describe in later chapters, managers refrain from micromanaging a project on a technical level but still implement various strategies to maintain control of a project's completion time. Developers also often give up on achieving what they promised and settle for a good-enough project in a good-enough time frame.

Another issue with software, as Brooks explains, is that it functions on a logic of constant improvement: nobody gets it right the first time, and often "one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time" (1975, 116). In programming, for example, programmers iterate a project by building one version, only to improve upon it in a second version, only to improve upon this in a third version, and so on. This means that no software project is ever complete, with each version being just good enough for the time being, to be improved upon in the following version.

The second type of good enough is good enoughness in corporate software work. After a few years of studying how corporate software developers build a seemingly boring everyday software product, I noticed that contrary to corporate discourses of efficiency, productivity, and meritocracy that permeate the corporate office, workers, most of the time, are doing work that's good enough and are happy with jobs that are good enough.

The two types of good enoughs do not function separately but co-inform each other: the good-enough worker in good-enough work conditions makes good-enough software. We can also flip this relationship around: if software has limitations to what it can do (be merely good enough), then a worker will settle for doing a good-enough job and come to work with a good-enough work ethic.

While good enoughness might superficially function in the excellence and efficiency discourse as something subpar or even as a failure, it can be embraced and accepted as something "okay." Good enoughness is about being pragmatic or realistic about the amount of work developers want to put into their projects and about the limitations of what a piece of software can do.

That said, good enoughness—particularly in terms of a good-enough work practice—can often be achieved only from a position of worker and company privilege. The worker who gets away with doing a good-enough job is a privileged worker. Good-enough jobs are sought after and coveted and often flourish in a culture that provides safe working environments. Not many software developers in an outsourced coding farm in Krakow or Bangalore, working to meet deadlines and concerned about their job security, would be able to work in a good-enough job (see Amrute 2016, 103). The same can be said for software. Only companies that were successful at building a software asset—meaning a product that continuously makes money—can settle into being good enough. Large old tech companies like Google or Facebook or even MiddleTech have certain assets (the search algorithm, the advertising infrastructure, the mapping engine) that they created years ago but still generate profit. Because they were eager, driven, and efficient years ago, these companies now have assets that give them the financial stability to be good enough in the present. A small start-up wanting to burst out into the tech scene and get noticed can't hire good enough workers and expect to financially survive. I'll discuss this dynamic in more detail in the next chapter but mention it briefly now to illustrate the "privilege of good enough." Being a good-enough company like MiddleTech means also supporting an inequality in work speeds and demands, allowing some people to sit back

and opt out of hyperproductivity while cruising on the unrecognized labor of other software developers and service workers.

This book is about a specific type of software worker in a certain kind of software company. MiddleTech is a specific type of company—one that sits on a certain software asset that allows it to be continuously relevant in a global software market. The company has a decades-old technology that is still embedded in various networks of software devices. Both the age and scope of MiddleTech are important for understanding how good enoughness emerges and becomes stabilized in such a company's culture.

Book Structure

This book's specific case study at MiddleTech brings to the fore a central mechanism in all software engineering, whether in Bangalore, Berlin, or Silicon Valley: that software is always merely good enough, in particular in companies sitting on older, still-valuable software assets. Like software's different layers of abstraction, this book is also structured in layers. Each chapter brings the reader into a different layer of abstraction that contributes to the larger picture of how good-enough software is made and good-enough work cultures are constituted. I begin with how programmers relate to their software, then move on to those who build software, and finally to the levels of management and organization that influence them.

Each of the following chapters addresses good enoughness in its own way and is structured around stories from my field. I take ethnographic storytelling seriously as I believe "stories display, juxtapose, figure, guide, and enliven in ways that philosophical concepts or abstract procedures cannot" (Kelty 2019, 4). While stories are too often dismissed as "'illustration" or 'evocation,' as if they lacked the (masculine) rigor of the 'concept' or the 'procedure'; stories . . . are the space of emotion and affect—too often demoted in power as something incidental, soft, solipsistic, not academic, or inadequately precise for thinking" (4). The first chapters will be largely based on the stories I encountered in my field, and the final chapter will be mainly analytical, focusing on the practices and figurations we encountered at MiddleTech.

In chapter 1, "Welcome to MiddleTech," I introduce the company, what makes it distinct but also similar to other "Medium Tech" software companies, and how this particular corporate software environment is the ideal site where good enoughness takes root and flourishes. I situate MiddleTech within the global software industry and show how its workers self-consciously

define themselves in opposition to Silicon Valley discourses, particularly through how they work. I highlight the many similarities between what I call Medium Tech and Big Tech companies, particularly in how programming work is defined, how management is organized, and how various management methodologies are implemented. I also explain how good enoughness flourishes in older companies (both Medium Tech and Big Tech) because their software is still embedded in various social and technical infrastructures currently in use—and making money—today. This dependency on an older asset turns the focus of a Medium Tech company to maintenance and repair rather than "disruptive" innovation.

Once we get a picture of the way in which MiddleTech is situated in the software industry, I'll focus on the software developers and their relationship with their community and technical objects. In chapter 2, "Software's Sociality," we get to know Ori, the Java developer-turned-lead software engineer, who helps readers imagine the type of care and compromise that programmers must constantly negotiate when building software. This is where the reader first encounters good enough at work. I explore the craft of working on software, showing how it requires the knowledge of the inner workings of a software system, experiencing moments of "closeness to the machine" (Ullman 1997, 40) and zoning in to a software environment to find a sense of flow in one's work. These ideal forms of care are often disrupted by various social and technical factors, and developers are forced to compromise and settle for something that's merely good enough for a customer to use. Describing software's sociality from the get-go is important as it helps the reader understand what is at stake and what kind of care and compromise programmers have to negotiate with their managers and customers when building software.

Focusing on yet another layer of abstraction, I bring us deeper into the social and technical conflicts that arise when working on software. Chapter 3, "Where Stuff Goes Wrong" builds on the understanding that software is a social object and paints a picture of the chaos, conflict, and misunderstanding that software inherently holds. I will show how conflict and controversy are inherent and inescapable in the software development process and an important part of understanding software development culture. I also frame the software company as a sort of "organized anarchy" (Cohen, March, and Olsen 1972), where the company's purpose or what it's working on becomes unclear for those working within it. To connect us to my central concept of good enough, I show that when stuff goes wrong, software is shipped to its customers in a state of good enoughness. While it may seem that stuff goes

wrong in any company, the difference with software lies in the rapid speed of change within the software industry, which is rooted in software update culture. The constant drive to update, fix, and innovate software means that it quickly becomes obsolete, and how it is programmed does too. This speed of change during software development challenges the stability of the knowledge of the people involved. These heterogeneous forms of knowledge result in processes of explanation and translation. Through explanation and translation between software developers, their code, managers, and customers, misunderstandings happen, and software development plans fall through the gaps between states of knowing and not knowing. Chapter 3 will also explain the different roles in programming, the nature of the customer-programmer relationship, as well as the role of management in organizing software work.

After describing how good enoughness is fostered through programming practices on an individual as well as collective level, I will introduce the processes of production and management in software development. Chapter 4, "Managing Good Enoughness," highlights how good enoughness in software work and the product results from the politics behind its development—both the macropolitics from the perspective of the software industry and the micropolitics from the perspective of the developer.

As Gideon Kunda showed, managerial ideology and managerial action designed to impose a role on individuals are normative demands that play out differently in action (1992, 21). To illustrate this, chapter 4 will outline the tensions among developers, their managers, and their machines, as well as how power and control are exerted, performed, and achieved when building software. While these forms of politics and power might be similar to those in other large corporations, my ethnographic descriptions underline the specificities of corporate software development, as well as the way in which power and politics influence how software is built, deployed, and how robust it becomes. Moreover, I also ethnographically show that software's materiality shapes the way in which programmers, managers, and customers interact with one another.

Chapter 4 also describes the deep tension between managers, who need to quantify their developers' work, and developers, whose goal is to build and fix their software, preferably with ample amounts of time. To highlight this tension, I describe the culture of speed and the drive for efficiency, velocity, or agility, which are all part of the office discourse at MiddleTech. I also describe the industry-wide software development management tools or methodologies that help drive this discourse (that is, the Scrum or Agile methodologies of organizing software work) and how good enoughness

becomes a way of pushing back against the desired outcomes that such methodologies aim to foster.

While my ethnographic stories are often more focused on the social and cultural dimensions of building software, in Chapter 5, "Slowdown," I focus more specifically on the culture of speed and efficiency when building routing and navigation software. Mobility systems, and the development of software for them, are intrinsically dynamic processes encompassing various temporalities, which are shaped by the interaction of sociality and technology. Yet slowdown is often at the core of software work. The slowdowns do not happen because the programmer chooses to take time to think through a topic; instead, slowdowns are imposed on programmers and their teams through various social and technical constraints. Once faced with these constraints, programmers need to compromise on what they are creating and releasing to the public. These slowdowns lead developers to create good-enough code. In chapter 5, I show how slowdown is the precursor to good enoughness, where part of a programmer's practice is halting the inertia of acceleration in the corporate software environment. Through various stories, we will witness good enoughness at work with constant stutters, blockages, breakdowns, moments of slowness, and deviations from the plan.

I conclude my journey through MiddleTech by theorizing the stories we encountered and placing them into a wider understanding of what good enoughness is and how it functions. To do so, I analytically explore good enoughness from a variety of angles, showing how different relational *constellations* inform good enoughness. Through this notion, we will start to understand the myriad of actors relating to one another and helping shape what "good for what" and "good enough for whom" can mean. When exploring the various stories of good enoughness in the previous chapters, we encountered different good enoughs for the programmer or good enoughs for MiddleTech's management or their customers. These parties have different concepts of what counts as good enough, which are often in conflict with each other and in need of negotiation. Of course this leads to compromise on what's good enough for the different parties involved. I will conclude by exploring the ways good enoughness is under threat, mainly by the forces of postindustrial capitalism that work against its logic, and how it is then kept alive.

This book is about the collective struggle to keep the software we all use alive, viable, and functioning. It is also a story about what is happening to our tech companies today, particularly the larger, older, aging software companies that built a good product sometime in the mid-2000s and are now

trying to maintain the one or two software assets that keep their revenue flowing. I paint a picture of one specific "software world," bringing you closer to places where software is made and maintained, while introducing you to the people who build it. I hope that this approach will also help personalize your everyday digital objects, giving you an intimate picture of software's complexity. I hope it will be good enough.

INDEX

Note: Page numbers followed by an 'f' refer to figures.
Page numbers followed by an 'n' refer to notes.