Speeding up, slowing down, breaking down: an ethnography of software-driven mobility

Paula Bialski

To cite this article: Paula Bialski (2020): Speeding up, slowing down, breaking down: an ethnography of software-driven mobility, Mobilities, DOI: 10.1080/17450101.2020.1816035

To link to this article: https://doi.org/10.1080/17450101.2020.1816035

Published online: 05 Oct 2020.

Submit your article to this journal

Article views: 9

View related articles

View Crossmark data
Speeding up, slowing down, breaking down: an ethnography of software-driven mobility

Paula Bialski

University of St Gallen, St Gallen, Switzerland

ABSTRACT

The dynamics of software – and thus also of its development – is an inherent part of the story of how mobilities are made and work. Building on this argument, this ethnographic study explains how navigation software development is caught in a constant culture of acceleration through commercial competition and shifting transport conditions, on the one hand, and a logic of routing and navigation based on creating the fastest route possible for its users, on the other hand. Behind this overall process of technological acceleration lies a multiplicity of forces – sometimes accelerating, but at other times slowing down, stuttering, moving in reverse, or completely coming to a halt in breakdown, shifting the pace of such technological progress. Bringing software development practice into the picture of how mobility systems work (or don’t work) allows us to understand the multiple temporal orders of speeding up and slowing down that push and pull at the fabric of our mobile infrastructures. Doing so will help us counter the popular discourse that our networked, seamless digital technologies are invincible. Based on an ethnography at ‘BerlinTech’, a large commercial navigation software company, this paper provides a multilayered understanding about the temporal forces fuelling our software driven mobility infrastructures.

BerlinTech is a large software company housed in a modern building with tinted glass windows in the middle of Germany’s capital. From the software developer’s desks, one can hear the city’s yellow tramway cars, bike riders and commuters beckon from the ground below – filling the room with light noise, almost reminding the engineers that they are building mapping and location software. Throughout a two-year period starting in July 2016, I spent 6 intensive months, as well as around 30 additional field visits observing these software developers make mobility software happen. These were everyday software engineers. Not the hackers with hoodies, nor the 120-hour work week Silicon Valley start-up stars. My field was filled with a quite ordinary, average group of people building digital maps: providing location data and other services to individuals (in the form of a navigation app on your phone or in your car) and other companies (in the form of location data information needed for building certain software). BerlinTech was a large software company, with around 1000 employees in its office in Berlin, and another 7000+ working around the globe.

The dynamics of software and its development is an inherent part of the story of how mobilities are made and function. Building on this argument, this paper will explore two points: firstly, that software-driven mobility systems are caught in a constant culture of acceleration enforced by two key drivers – commercial competition in the software industry, as well as the inherent logic of smart routing and navigation that is based on creating the fastest route possible for its users. These two
factors mutually enforce a culture of acceleration: as software market competition builds better software, transport conditions attempt to be faster and more fluid, feeding data about their transport conditions back into the software that then attempts to improve, and so on and so on. That being said, I aim to show that this interrelated mutually-enforced culture of acceleration does not – contrary to many hopes and dreams of the actors involved – create a constant improvement in the efficiency of movement or software innovation. Instead, there are constant moments of slowness, stutters, blockages, breakdowns and nothing ever really going to plan.

While it might seem that this dynamic within the navigation software development culture might lead to a temporal crisis, whereby constant acceleration of innovation (e.g. of digital navigation software) and of mobility together will run more and more into unpredictable and unfixable hurdles, I suggest a more complex picture of our software-driven mobility. This approach does not focus on acceleration – but rather on the constant relationship between a multiplicity of forces of acceleration, slowness, and breakdown together. Doing so will provide a more realistic picture of the complex fields of temporal forces which are mutually reinforcing each other (and sometimes not) in our software-driven mobility systems.

In short, bringing software development practice into the picture of how contemporary mobilities and mobility systems work (or don’t work) allows us to understand the multiple temporal orders of speeding up and slowing down that push and pull at the fabric of our mobile infrastructures. Doing so will help us counter the popular discourse that our smart cities or our networked, seamless digital technologies can help speed up our sluggish old real-world infrastructures, that transport can be made more quick and seamless or that capital can flow faster. As I will show in this article, both physical and digital infrastructures are both static and dynamic at the same time, highly entangled in sociality, and adhere to and produce specific temporal cultures of production that shape how mobilities happen. Through an ethnographic case study of navigation and mapping engineers at BerlinTech, I will explain the culture of acceleration, and the resulting forces of slowing down and breakdown, that are a key part of how mobility is made.

Methods of studying software developers

My story at BerlinTech began in June of 2016, through a friend and door-opener named Ori – who had worked at the company for a few years prior to my first visit. He somewhat reluctantly put me in touch with his managers, who took a bold (and confused) leap in accepting an ethnographer into their team – something that never happened before at their company. The team consisted of 100 developers who built a map-based app similar to any navigation app that sits in your phone. I spent July, August, and September with the ‘front-end’ team, going into work on a daily basis with the rest of the developers. I would come into work at around 10am and leave at 5 or 6pm.

My presence at BerlinTech was ‘official’ in the sense that I met with the HR department before starting my fieldwork, who noted down all my details, gave me a keycard to access the building, and provided me with the title of ‘visiting researcher.’ My position was unpaid, although I had a supervisor (a manager of the front-end team at the time) who was responsible for me at the company, and directly informed his employees of my presence. I received a desk in the middle of the large open office. Before entering the field I enrolled in an online beginners course in computer science in order to understand the very basics of programming. This gave me a rough understanding of how the practice of coding worked, but couldn’t program anything myself. During this period, I attended their work meetings and listened in on their conversations. I kept my observations in a field diary, recording conversations and thoughts, which I would spend elaborating on throughout the day or in the evening after finishing my day’s fieldwork. After a few weeks all 100 developers understood what I was doing there, and would invite me to meetings or for one-on-one coffee breaks to explain what they were working on.

Throughout the fall, winter and spring of 2016 and 2017, I would sporadically travel to Berlin and keep meeting individual informants for lunch or after-work beers whenever they had time. During
July, August and September of 2017, I returned to more intensive fieldwork with a completely different team in BerlinTech. The ‘Routing and Navigation team’ was again composed of around 100 developers, and would help give me another perspective of how developers worked. This group were known in computer-speak as the ‘backend team,’ meaning the group working ‘closer to the machine’ on parts that the regular user of the navigation app doesn’t see at all such as running the map operating system, working with cartographic data, choosing and optimising the right routing algorithm, etc. That summer, I would come into work at around 10am and leave at 6pm, and I received access to the BerlinTech internal software system and was able to observe the entire communication of the company – including various posts and message boards in their internal social network, or their more technical notes which were shared within a Wikipedia-like internal site.

Much like the summer before, I allowed my field to dictate my contact-hours, and I would often find myself at concerts or festivals on Saturdays with a small team, or after-work happy-hour with another group. One key informant lived in my district in Berlin and I would walk home with him nearly every day. I continued this fieldwork throughout the winter of 2017 and 2018, sporadically coming into Berlin around twice a month for follow-up interviews or meetings. Like many long-term ethnographies, my time spent at BerlinTech was not structured from the start, and seemed to grow into various directions during the process of fieldwork – allowing the field to dictate with whom I spoke to, in what context, and at what frequency.

My ethnography, inspired by my colleagues and teachers situated in more anthropological research, was a slow, very intimate ethnography, which at some point became ‘totalizing,’ drawing upon the ‘whole being,’ and not reducible to the ‘collection of data’ (Oakeley and Callaway 1992, 3). This experience gave me a more holistic perspective of the software development process – as one cannot understand the full complexity of software development without understanding, in some way, both front-end and back-end programming work. I wanted to understand what it really meant to be close to a machine, and to intimately understand how to build software. Between my intensive summer research stays, and until the summer of 2018, I would connect to my informants on a monthly basis – either in person, or via email or chat. Many of my informants became my friends, and the borders of personal life and formal ethnography began to blur and the ‘research “field” lost its boundedness’ (McLean and Liebing 2008, 23.). The names I used in this article – including the name of the company itself – are all pseudonyms. The information I gathered about the conversations and events that happened at BerlinTech were all collected with the employees’ consent, with the knowledge that I would publish my findings in academic journals or monographs. The quotes and anecdotes I used in this article stemmed from transcriptions from my field diaries.

**Software for mobilities**

I started studying corporate software developers in the first place in order to uncover that which countless anthropologists, science and technology scholars and authors have done before me: how the human, machine, code, and culture mutually influence one another (Coleman 2012, 2014; Kelty 2008; Kidder 2011; Mackenzie 2006). I was interested in the tensions, the negotiations, the intimacies and the politics that bubbled up from their interactions. These two years, among other things, taught me that our software is not just a set of commands put together and the systemic relations arising out of these commands. Rather, as Nathan Ensmenger underlined, software is history, organization, and social relationships made tangible (Ensmenger 2012), where ‘values, opinions and rhetoric are frozen into code’ (Bowker and Star 2000, 35). A piece of software holds decades of development and design, ways of organizing, and ways of working and being together. Through understanding software development culture, we can also come closer to understanding intimately how our everyday digital infrastructures are built and maintained, both in their limitations and possibilities.

Software is not just important, but is important for mobilities. The modern city – and perhaps our transport infrastructure ‘exists as a haze of software instructions’ (Amin and Thrift 2002, 125). Over 10 years ago, John Urry outlined that the ‘mobility turn’ connects the analysis of different forms of
travel, transport and communications with the multiple ways in which economic and social life is performed and organized through time and across various spaces (Urry 2007, 6). Since then, mobilities scholars have underlined the way in which these mobility forms are being increasingly defined by new media technologies (for example tools like navigation software that mediate our mobility). New mobility systems today are much more interdependent with each other so that individual journeys or pieces of communication depend upon multiple systems – such as software systems that need to speak to each other in order that particular mobilities and ‘sortings’ take place – and these software systems all need to function and interface effectively with each other (Sheller and Urry 2006a, 7). These mobility systems are ‘complex, computerized and risky,’ (Sheller and Urry 2006b, 7) and yet are central to new urban forms and practices, and to the lives of city dwellers. Mobility systems are, in essence, run by media, and can be seen as media themselves in their process of transition and mediation (Schabacher 2013).

Graham was at the forefront of understanding this interdependency of physical space, mobility and code, stating that ‘flows, mobilities, and transactions; the folded geographies of inclusion and exclusion; the construction, consumption and experience of place … all, very literally, are now performed – at least in part – through the continuous agency of vast realms of computer software’ (Graham 2005, 4). This ‘networked urban mobility’ is experienced through software – suggesting a fundamental change in the everyday practice of mobility, now invisibly delegating our coordination to smart and intelligent environments (Freudendal-Pedersen and Kesselring 2017). Dodge and Kitchin rephrased space as ‘code/space’ – meaning space that is dependent on software-driven technologies to function as intended, such as check-in counters at airports or workplaces dependent on shared calendars (Kitchin and Dodge 2011). Other recent mobilities scholarship has also highlighted the entwined relationship between driving and navigation, and the role of software in pushing this desire for the optimization of our automobilities (Hind 2019 in particular looked at the role of autonomous vehicles and how their software is radically changing our norms of navigation). Elsewhere, Hönke and Cuesta-Fernandez (2018) showed how software, in particular logistical software, is one of many controversial actors that powerfully shapes how mobility is governed.

While navigation software like the one that sits in your car or in your smartphone doesn’t necessarily create a code/space (a road will still exist and be drivable when the navigation software breaks down), Dodge, Kitchin, and others I referred to underlined the power of software in reinforcing and deepening a certain spatial logic, delineating social action (where to act, drive, check-in, interact, etc.). Elsewhere, and quite early on, Graham and Wood termed this mechanism ‘software-sorting’ where they predicted that mobile technologies will make social and demographic information available in real-time, configuring real places over various temporal scales (Wood and Graham 2006).

As we can see, software today inherently drives and shapes our modern-day mobilities. This brings me to perhaps a rhetorical question: if we want to understand how mobilities are being configured by our socio-technical systems, why study just software? Built infrastructure is also a socio-technical system. Don’t roads or bridges also delineate social action, steering mobility? As many STS scholars have underlined, infrastructure and artifacts have politics, they influence us, they are inherent actors in our sociality (Latour 1990; Winner 1993). Indeed, one can imagine a bridge steering drivers or pedestrians one-way and not another, and the engineers behind the construction and maintenance of that given bridge as being responsible for mobilizing a certain direction of movement. Yet the interesting and perhaps quite unique thing that sets a software system apart from the built environment is that software is dynamic.

While indeed all infrastructures constantly adjust, adapt to, reshape, or even internalize elements of their environment in the process of growth and entrenchment (Edwards et al. 2009), as I will show later in this paper, software lives under a rapid and continuous temporal order of evolution and acceleration. It is an ‘object subject to continuous change and lived with over time as it evolves’ (Cohn 2019, 423), one that does not sit still ‘long enough to be easily assigned to conventional explanatory categories’ (Mackenzie 2006, 18). Software is no longer sold in a shrink-wrapped box,
ready to be installed on a hard drive, never to be touched, tweaked or updated again. Software is currently a service (Kaldrack and Leeker 2015) where new monthly, weekly, daily or hourly software updates created by a team of software developers feed into your current software version through Internet-enabled updates. This creates a certain dynamism – where software is able to change, able to be updated every week, hour, or second of the day – with these updates going directly into the apps on your phone (such as BerlinTech’s navigation app). These updates, as I will show empirically, include completely new features, include updates to a feature or optimization or ‘fix’ of an algorithm within the map, or new live data about road conditions or traffic acquired from countless cities, towns or regions.

Additionally as I will show through my empirical examples, building and maintaining software infrastructure is quite different from maintaining a railroad track, bridge or freeway as it is not about wear and tear or the failure of particular bits of software. Rather it is about keeping software in sync with changes and dependencies made in other software and systems.

Software developers matter

This brings me to my next point. When addressing software, we must also shift our gaze towards the people who make it. ‘Software also refers to its producers. Someone or something codes it; there is an originator’ (McKeznie 2006, 14). Where considerable research has been done on hackers (Coleman 2012; 2014), free software activists (Kelty 2008), or game developers (O’Donnell 2014), much less has yet been written about the silent majority of computer specialists building large-scale corporate software, the ‘vast armies of largely anonymous engineers, analysts, and programmers who designed and constructed the complex systems that make possible our increasingly computerized society.’ (Ensminger 2012, 3). Doing so is crucial in understanding the close merger of software development, commercial pressure, and capitalist labour processes.

Shifting our attention to my field, I want to underline that software is not created by lone hoodie-wearing coders, but by a collective effort – created by sometimes hundreds of developers, working together in real time. One mistake of a developer on the 6th floor can affect the outcome of the developer’s feature on the 5th floor. Software systems are also not static because they include layers of economic and social orders of design, algorithmic logic, deadlines, and programmer creativity that all run on their own temporal orders, thus instigating, shaping, and slowing down mobility.

So how is a piece of navigation software made? BerlinTech bases its business on two types of customers: firstly, they provide map and navigation software for users, in the form of a desktop or smartphone app. Secondly, and perhaps their main focus, is providing maps for other businesses. This includes in-car navigation systems (mainly for German car companies) and on-demand ‘location services’ for anybody building a technical infrastructure like an app that needs a map or navigation component (i.e. imagine a restaurant recommendation app that needs a map feature to help customers find their Fish & Chips stand around the corner. That map embedded into this app could potentially be coming from BerlinTech).

This software is a complex navigation system, which is socially constructed using a variety of actors – the main technical players being the frontend and backend developers and their managers, the designers, as well as the cartographers gathering road data. Amongst others I don’t focus on here in this paper, but add to the community making this software are actors in upper management, the cleaning staff, the company legal team, door clerks, human resources, the privacy team, or the cafeteria staff.

If you imagine a navigation app installed into your smartphone, the components your eyes glance at and your thumb or index finger tap or swipe are made by a team of people called the ‘frontend’ developers. This team at Berlin Tech was made up of around 100 software developers, and these men and women created the components of the software the users interact with – so a screen, a browser, or an icon. They work in programming languages like HTMS, CSS, or Javascript that deal with the application layer of the software itself. Using another metaphor – if you imagine a software project
like a large house, these languages help build the walls, the windows, the paint colours – everything you see. The frontend developers never directly access the hydraulic system, heating system or scaffolding below the surface layer.

Work between developers and non-developers is constantly about translation and explanation – all of which takes time. It is also about carefully managing a ‘power imbalance’ – which has to do with a clash between developer’s norms and standards of how software should be built, and designers attempting to implement their own contrasting visions.

Backend developers, on the other hand, sometimes derogatively look at frontend developers as ‘pixel pushers’ – meaning those programmers who ‘merely’ interact with designers and tweak the dimensions of pixels around on a screen. Marcel, a backend developer, admitted that he dislikes mulling over the look of his website or app, or what the user will do, or how his application will be used in general. He preferred thinking through problems or the architecture of the whole system. He also admitted that the less talking to other developers, the better – and backend developers are stereotypically known for being bigger loners (fieldnotes, August 2016).

The backend developers do not work on everything, but are rather split into various roles with specific tasks needed to create functioning navigation software. For example, a navigation system bases its navigation on a map. This map has to come from somewhere. At BerlinTech, a team of cartographers gathers the core map data that the entire system is run on. These cartographers, database engineers, or geotechnicians add or update road, lane and other landscape information to the main BerlinTech database. Roads are also categorized based on a number of factors like the type of road (large highway road, or small country lane), or its incline. These engineers are dispersed across the globe and I had little contact with them during my fieldwork. Yet I did observe the work of a particular team – the ‘Route Quality’ engineers – that worked closely with the core map data. At times cartographers make mistakes and a road veers off where it shouldn’t, or is improperly tagged (ex. a large road is tagged as a gravel lane). The ‘Route Quality’ engineers literally ‘clean up’ inconsistent data, making sure that the ‘core map’ database truthfully represents the ‘real world.’

Additionally, the world does not stay the same – road closures, new roads, traffic jams and special events change the dynamics of a route. Thus, map data is not always static, but can be fed into a map system dynamically, in real-time. This dynamic data updates is acquired by BerlinTech from various cities, states or regions that provide this information to them at little or no cost.

Once the map data is collected and cleaned up, a routing team (backend developers) makes use of this map data, but then constructs another software layer that helps route a commuter or car from point A to point B. When routing a commuter, developers have to take into account if the driver wants to take the shortest route or the fastest route, which forces them to think about the road category, the amount of left turns on the route (left turns eat up more of the driver’s time than right turns!), traffic congestion, etc. The list can go on. At BerlinTech, the ‘routing engine’ was run on a ‘core routing algorithm’ which was implemented and maintained by a wise group of highly skilled computer scientists – many of them graduates and PhD dropouts from prestigious computer science departments. These specialized programmers spend a lot of time tweaking standard routing algorithms to fit their software (and their drivers’ needs).

By now you are hopefully getting a picture of how a navigation system is constructed. It involves mapping the world and recording it in the form of data other developers can use, making sure this core map data exists and is consistent, downloading dynamic traffic and road data from various regional providers, and implementing and tweaking a certain routing algorithm and making sure it works well with the map data. These developers have to interact constantly with each other – both directly in meetings and conversations – and indirectly through their code, as they are constantly reliant on code written by other team members. These development teams – both internally and between other developers in the company – have to agree on a set of rules and expectations, as well as the time it will take to complete their project. Doing so is not easy. Developers are forced to form coalitions, negotiate their creative decisions, agree and disagree and compromise and, most pertinent to our topic in this paper: they have to do so under a certain culture of speed and efficiency.
These moments of negotiation directly affect the software product, as they at times shape the actual outcome and the time in which a piece of software is completed. Thus looking just at software and not the producers will not give us the full picture of how a mobility system works.

**Mutual acceleration**

Up until this point, we discussed the importance of software in understanding mobilities today, as well as the importance of researching the people who build the software that help shape our mobilities. The next task is now about underlining the type of temporal culture under which this software is produced. The ‘temporal’ is understood here as a specific experience of time that is ‘structured in specific political and economic contexts’ not only a ‘uniform time tied to a particular technology but rather a time particular to the labor and other forms of social difference that produce them.’ (Sharma 2017, 133). As acceleration ‘became a key measure of progress’ in our modern societies (Wajcman and Dodd 2017, 2), and digital media technologies were linked to such progress, the software industry started to foster a culture of acceleration. Today ‘the expansion of our horizon of possibility’ becomes ‘an essential element of the promise of technology’ (Rosa 2013, xxxvi). Our sociotechnical imaginaries of innovation (Jasanoff and Kim 2015) envision digital media as fast, seamless, and highly reactive to change.

The topics of speed, efficiency, velocity, or agility were all explicitly part of office discourse at BerlinTech. Industry-wide software development management tools or ‘methodologies’ help drive this discourse. The most prominent within BerlinTech as well as throughout the global software industry is something called the ‘Scrum’ methodology. Scrum was introduced into the world of product development methodologies in 1986 by two Japanese professors of marketing, who were looking for a way to make product development faster and more flexible, reactive to the changing demands of the market. While Scrum was not originally intended for software development, the internet brought a change to the way in which software was built, deployed, and could be updated – making it very reactive to market demands. A need for ‘agility’ emerged. With this ‘agile turn,’ (Guerses and Van Hoboken 2018) the complexity, distribution and infrastructure of software changed – as well as the temporal orders of production. A new production order, characterised by short development cycles, continuous testing and greater simplicity of design (Douglas 2015) also attempted to speed up the developers work, and deliver to their customers and their users quickly. Scrum emphasizes that teams come up with solutions and meet customer requirements by self-organizing and communicating a lot, advocating adaptive planning, evolutionary development, early delivery, continual improvement, and it encourages ‘rapid and flexible response to change.’

While I won’t get into the strange and fascinating issues around Scrum in software development, it is important to mention it here as it is one of the central methods that temporally order software development work. Software development managers structure the developers’ production schedule by using the Scrum methodology, with software developers working in 2-week increments called ‘sprints’ in which they have to ‘deliver’ small changes, updates, bug fixes, or innovations within the software.

‘Sprinting’ is important for two reasons – both tied to market competition. Firstly, at BerlinTech and elsewhere, commercial software development is about building a product for somebody. BerlinTech resides within a service industry where software serves customers, meaning it is built based on the specifications of another company, which runs on their own market competition. For example, if a car manufacturer is building a new model of their car, they will want to get it out as quickly as possible, in order to provide a newer product for their users, giving them an edge over their competitors. The software that goes into that car must also be finished in that time. This product-oriented perspective inserts demands on the software development team – and these demands help organize deadlines, requirements, standards, and the type of components that needs to be completed for the customer.
Additionally, BerlinTech also competes with other digital map companies, and wants to provide a better, faster, more robust, less-buggy map that would win over new users, or retain their existing users. Both this customer-driven and internal market competition enforces a constant acceleration, a constant improvement in efficiency, and a constant demand to build software faster.

At BerlinTech, this efficiency is measured in something called a ‘KPM’, or ‘key performance metric’ – which is a term used by project managers around the world in a variety of industries to measure the level of output of production. A KPM in software development might be the amount of features completed on a certain day (the more, of course, the better!), or the amount of bugs found in a piece of software (the lower of course the better!). An upper manager, standing in front of a screen of numbers in a meeting at BerlinTech, explained to his team during a ‘touchdown’ meeting:

‘As you will notice – there is a lot of focus on the KPM resolution till week 44, all our work streams have good projections. Yesterday (the team leads) and I have updated the numbers. I think what we should keep in mind that there will be a continuous struggle with (our new software product). I think we need to think of how we are addressing the velocity of bugs. We are not necessarily fixing them. This is the number of bugs per week going from 2 up to 2.2. In the Rendering team, their velocity is 5.’ (fieldnotes, August 2018).

From this short excerpt, we can see the culture of acceleration in action. The manager uses terms like ‘velocity’ – which is used to indicate the speed of progress within a team – a further metaphor delineating temporality of their project. In this instance, a high velocity indicates the amount of bugs that were fixed in the shortest amount of time (from the moment the bug was noticed by a customer to its repair). The culture of acceleration is further fuelled by competition with other teams. Here, this manager is giving his developers an example of the Rendering team who achieved a higher velocity in fixing bugs than the team to which he was speaking.

KPMs and scrum-related methods of production attempt to create ‘rapid’ and ‘reactive’ forms of software delivery, and a certain commercially-driven narrative: that software has to be produced quickly, and that speed is of utmost importance in order to beat competition. A developer named Timon, during a feedback meeting to his management and team, explained:

‘I think we should stop our urgency mode. We have this urgency that keeps coming back. We need to fix bugs, we need to deliver, we need features delivered we haven’t even started. I know we are working on products better, but I don’t think we are in balance right now. Look, we just started our project now, and we are already over the deadline. I’m not saying this is his or his fault. I am saying this is the company.’ (fieldnotes, August 2017).

One of the tools fuelling ‘urgency mode’ are the large display screens – 4 of them on each floor – that display a running slide-show of company-wide messages to their employees. In between sparkling photos of the world’s smart cities, night-time photos of highways lighting up in gold from the low shutter speed of the camera, is a display of the amount of bugs that are being detected in each individual software product that BerlinTech makes. Analysts were increasingly being used within BerlinTech to optimize the developer’s work, and to show them what they are doing wrong – based on various variables like the amounts of bugs being detected or user satisfaction.

These screens are one of the tools fuelling this ‘urgency mode’ that ‘keeps coming back’ – and is symptomatic of a commercial software company’s culture of acceleration. This culture of acceleration becomes a constant regime under which developers should produce software.

**Slowing down with software**

The reality is much more messy of course. While developers like Timon do notice and complain about urgency mode, in practice there is a large disconnection between the demands of a certain management team or customer, and the reality of how software actually works and is built. In the following section I will explain how the way of working with software is prone to slowness. I will also explain how the very material constitution of software makes it prone to breakdown.
During one of my lunch-hour interviews at BerlinTech, Andrew began to explain that routing algorithms are used in various mapping technologies – not only in navigation systems, but also for software that tracks fleets of trucks or boats. Andrew was part of the ‘core algorithm team’ – a PhD computer science dropout who specialized in routing algorithms. Routing at first seems quite simple – you have to get a car from A to B in the shortest time. But Andrew explained that the challenge is the size of the data you have to analyse to reach your destination. There are millions of streets in the world. And many possibilities. So in order to calculate a route, a programmer like Andrew has to make certain decisions and reduce the amount of data to analyze in order to calculate a route. He explained: ‘so maybe you also have to introduce heuristics’ – meaning certain algorithms that makes an intuitive sense and chooses the right way to go based on certain factors. ‘And then you end up with an algorithm which defines a “best route” value’ he explained.

‘But how do you define “best” route? Isn’t “best” quite different from car to car, and from driver to driver?’ I asked. ‘Well that’s a good point – we do assign a cost to every maneuver, to every road. When there is a right turn, that’s a cost, when it’s a left turn, that’s another cost. You sum up travel time as well as length of a road. And every company who builds this system does things differently. So our product might calculate costs a certain way, according to a certain logic we came up with on how to enhance a route quality, and other company might do things differently.’ ‘So why are there a lot of different algorithms used for routing?’ I asked again. ‘Well there are a lot of tradeoffs involved. And you have to calculate for these tradeoffs. For example, what if you were driving on a route and you had to catch a ferry? You can a) drive quickly, use a lot of fuel, but you still make the ferry on time, or b) drive slowly but miss the ferry. Which do you choose? And what if you suddenly get new ferry route data? You can start to imagine that there is a constant need for optimization.’ I asked Andrew what factors go into optimizing a route. He explained that there is a) traffic, which includes dynamic traffic data (so stuff that changes in ‘real time’), and historical traffic data, which can help with predictions of traffic (you have a record of the typical traffic congestion at a certain time of day). You can also have b) static conditions, like ferry departure and arrival times, and c) you can have certain streets where certain cases of driving are forbidden at a certain time of day.

Andrew gave me an example of Munich, which changes the driving direction of streets during rush hour. ‘Then there are other regional factors like d) express lanes, which typically exist in America’ (fieldnotes, August 2017). The list of variables, Andrew explained, are endless. The practice of optimization of an algorithm is about including or adding certain parameters and excluding others. Optimization can result from merely new data being available (like new ferry timetables), or a car feature (e.g. If a navigation algorithm takes into account a gas or diesel car’s fuel consumption, the algorithm will have to be optimized for electric vehicles which require charging stations along a route). New technologies of collecting map, navigation, or route data change the type of data that is available to Andrew and his colleagues.

Sergei, another developer explained, ‘in the end the goal (for the customer) is always the same: finding a more optimized, better route. If you find something better … you go back and technically investigate this, and say “sorry guys we have 3 solutions possible”’ (fieldnotes, August 2017).

This subjective temporal estimate merely performs speediness and efficiency. The process of optimizing an algorithm, or ‘calculating tradeoffs’ as Andrew explained, is not an instant process but takes time. Andrew is dependent on his team, on the type of algorithm they choose, as well as working with the complex data they already have. Additionally, as Andrew pointed out, they can ‘suddenly’ get new ferry route data – which will have to be implemented into their optimization approach. Andrew and his team are thus dependent on each other, their existing software infrastructure, their own data they collect, as well as third-party information from town, cities, or regions. All these pieces are quite dynamic and cannot be calculated into rough ‘man months.’ As Sandra, another backend developer explained: ‘how could I assess something I don’t even know how to build yet?’ (fieldnotes, August, 2017). The method of building, optimizing, or tweaking something often arises while building it, and not beforehand. The process is messy and full of improvisation (Feyerabend 1993).
Competing visions

And...
interdependency of software developers’ work, and what happens when they reshuffle their team. He replied: ‘It’s a catastrophe here. I am always afraid if somebody goes on vacation or another person leaves the company. This is a real issue here’ (fieldnotes August 2017). Software developers’ own personal temporal order (their vacation days, gaining another job and leaving in 2 months) often influences the way in which code is built. While planning and optimising their system, I noticed that some leave for vacation, or go on paternity leave, or move teams, while others leave the company together. All sorts of social and organisational issues – the mess of everyday life – can slow down or speed up the planning of software, in turn affecting how this software will bring us from A to B during our afternoon car ride.

**Legacy**

Another factor that slows down a software project is software legacy. Developers don’t just write new code every time a customer wants new software. Software has a history – or a legacy. Legacy code is code that has a wide range of definitions and origins: it is sometimes referred to code that was acquired from another company; and other times it refers to code that was left behind by people on the original team, who moved on to other projects or found other jobs in different companies. Legacy is also inherent in the language of a system used in the past, that confines a programmer to work with the language in the present, limiting their ability to learn and move forward.

Legacy causes infrastructural decay: ‘it is composed of multiple lifetimes of different parts of the system – hardware, software, code, organizational processes, programming languages, institutions, careers – all of which are entangled and are aging or obsolescing at different rates’ (Cohn 2016, 1513). This infrastructural decay is at times highly frustrating for a developer, as it makes them feel like they cannot move forward, as the code they are faced with places limitations on what they can build on top of it (for example, by ignoring the legacy code, or not cleaning it up or ‘refactoring it,’ the developers’ software might collapse altogether). At other times, legacy code – much like a road that has already been paved once – makes software development much more predictable for the developer, providing more stable foundations, avoiding crashes and breakdowns. If you speak to any software developer, they will tell you that legacy determines how they work. It also shapes or limits their creativity – what they can and cannot do with their code.

Fernando, a developer in the frontend team, explained: ‘It’s very tempting that if a project fails you just forget about it. You don’t delete from the source code.’ (fieldnotes, August 2017). This ‘temptation’ is sometimes due to the developer’s feeling of a limitless memory system that can store all their code for all eternity. As computer engineer Bill Blunden explains, ‘Engineers in the days of yore had to meticulously balance memory usage with the need to minimize the number of CPU cycles consumed … Engineers today are not faced with the same pressures to squeeze every ounce of performance out of their programs’ (Blunden 2008, xvii). What Blunden here means is that due to improvements in microchip storage capacity, computing power has gotten faster. Memory capacity due to cloud computing solutions is now limitless. So software developers do not have to worry about having too many lines of code – as long as their software works, and the performance of their software is up to speed, a lot of code can be lying around in the code stack.

One of the main reasons for leaving code lying around is, as another developer explained, due to the lack of trust in your new system: ‘Well you are not sure your new way will work. So you just keep it. Because you might want to switch back to your old version at another time. Its like a safety net’ (fieldnotes, August 2017).

The amount of legacy of a system has a direct impact on how software gets built. Older companies are bound to have a large stack of code (called a ‘legacy stack’) piled up under them, often using older programming languages and older programming methods and foreign design choices. As one of my informants explained, these companies have a hard time ‘innovating’ and moving with the demands of the technology market. Moreover, the process of ‘refactoring’ –
adapting older code to fit a new system – takes time. Developers feel like they are mind-reading, trying to understand the logical order the other programmers who came before them were working with. Software at BerlinTech is thus not built by one team, but by a slew of teams over time, with layers of design decisions and ideas of how routing and navigation should work. To understand how legacy directly influences the mobility of a driver, I will take another example from the field. One backend team working on voice guidance within the in-car navigation system was discussing building a new feature that would update left hand turns:

“Giovanni: I would make this feature first, and then adapt the rest. So adapting the legacy in guidance to the new code base.

John: Just to be clear, there is no work happening on legacy guidance, right?

Giovanni: Legacy guidance has to be cleared, so no there is nothing.” (fieldnotes August 2017).

In this discussion, the two want to build a new feature that relies on cleaning, or ‘refactoring’ some legacy code written a few years before to ensure that the new feature will work smoothly. I see Giovanni and John’s discussion as being typical wishful thinking: developers can and do make features without adapting the old, rusty legacy code base. These features then break down, steering developers in the wrong direction. In another instance, a code base became such a mix of new and old legacy code that drivers were sent on a ‘spaghetti route’ (fieldnotes July 2017), meaning loops over bridges, outside ramps, and down circles (making a driving path look like spaghetti, hence the name). When not dealt with (and it is often not at BerlinTech), legacy code can create a mess of bugs, resulting in the wrong routing directions or faulty lane data that changes the shape and route of a driver.

A myriad of social, political, economic factors help uphold infrastructure, shift it, break it down, or shape its future development. Infrastructures are thus determined by a specific temporal order that determine their development, upkeep and demise. One such temporal characteristic of infrastructures stems out of its historical depth – or legacy. An infrastructure is not something you build anew – but is something that holds a legacy, as new train tracks are layered in place of train tracks that came before them, streets are built on other streets, cables are built where other cables are laid, and so forth. Current software infrastructure thus ‘wrestles with the inertia of the installed base and inherits strengths and limitations from that base’ (Star 1999, 382).

Legacy software holds another temporal and normative characteristic: it needs to be patched, fixed, or mended – as it constantly breaks down. These moments of fixing can take time – slowing or speeding down mobilities in and through the given infrastructure. This ‘work of keeping software present (maintaining its currency, knowability, relevance)’ (Cohn 2019, 426) highlights the constant tension between the new speedy innovation driving software and the old software that constantly has to be brought up-to-speed so-to-speak. It is not just the software worker’s sociality, their multiple moments of competition and negotiation on how to optimize a feature that slows them down, or their dependence and synchronization with live ‘real-world’ traffic systems, but also the interaction with ‘software’s lived durations’ (Cohn 2019, 426).

**Rethinking temporal forces**

By this point, we start to gain a picture of two different forms of temporality inherent in software-driven mobility systems: 1) the speed of mobility (in, for example, the optimization of traffic) and 2) the speed of software development, driven by consumer-oriented market competition and imaginaries as well as real sociotechnical innovation.

In theory, both of these forms of temporality mutually-reinforce a state of acceleration. What I mean here, is that as better software is produced in competition with other software companies, the more synchronized the software should theoretically be with our road infrastructure and traffic data, and the faster the user should be routed to a given destination. With each next software iteration, the route should become even faster, the traffic should become
even more seamless, and so forth. This is ‘intentional, goal-directed processes of transport, communication, and production that can be defined as technological acceleration’ (Rosa 2013, 82). Although it is not always easy to measure the average speed of these processes, Rosa explains that the general tendency to accelerate in the realm of technology is undeniable (Rosa 2013, 82).

However, behind this overall process of technological acceleration lies a multiplicity of forces – sometimes accelerating, but at other times slowing down, stuttering, moving in reverse, or completely coming to a halt in breakdown, shifting the pace of such technological progress. And all this often happens simultaneously and on different scales. Knowing this gives us a picture of a plural landscape of temporalities – shaped by the interaction of sociality and technology in-action.

As I have shown in this paper, both of these forms of temporality inherent in software-driven mobility systems are also driven by these other temporal forces. When it comes to mobilities – we know that slowness happens. As one of the programmers at BerlinTech mentioned, there are moments where traffic has to wait for ferries to arrive, for bridges to open, or for traffic to get decongested. There are also moments of breakdown – when car crashes or traffic jams happen – often, but not always, due to the excess of speed. In software development moments of negotiation, explanation and translation between developers can take time, slowing down the process of innovation or implementation. Legacy code – or years and years of code that makes up the underbelly of a given software project – also takes figuring out and refactoring, which also all takes time. Additionally, many of these moments of figuring out, negotiation, and interaction can lead to human error, which does make software buggy and breakdown. As I also highlighted, large software projects, with layers of legacy code, create entanglements and ‘spaghetti code,’ deeming the very material constitution of software very prone to breakdown.

That being said, perhaps a more realistic picture of software-driven mobility systems is to imagine them as moving at various temporal forces at once, in an entangled triad of acceleration, slowness, and breakdown. These various temporal modes happens where one entity (like a navigation software) works at various tendencies, at moments accelerating, and at other moments slowing down, and yet again breaking down. My contribution to understanding software-driven mobility thus expands our understanding of such systems as driven by continuous acceleration, and takes a more complex approach, offering an understanding that forces us to account for a combination of factors like slowdown, breakdown, as well as acceleration.

Doing so also avoids scandalizing such acceleration, and creating a worried hysteria that yet another software crisis, and perhaps even a mobility crisis, is looming. In the past, references to the chronic ‘software crisis’ were ubiquitous (Ensmenger and Aspray 2002) within popular discourse and throughout the industry. These so-called ‘software crises’ stemming from the acceleration of software production and its too-hasty evolution for the infrastructure that can contain it. For example, in the 1960s, there were ‘scandals, complaints, laments, and self-recriminations’ the software community that an industry-wide crisis was imminent (Ensmenger 2012, 24) due to the fact that programming was not seen as a discipline, nobody was training professional programmers, and computing innovation was moving too quickly causing a huge shortage of workers (Ensmenger 2012). The 1968 North Atlantic Treaty Organization (NATO) Conference on Software Engineering attempted to set an agenda that would solve the perceived shortage of a ‘certain type of qualified programmer’ (Ensmenger 2012, 24). This rapid speed of innovation was driven by Moore’s Law – as ‘computing power, as measured by the density of the silicon chips that went into these machines, was doubling about every eighteen months’ (Ceruzzi 2003, 297). There have been many other moments in popular discourse that warned us that a total crisis of speed is looming. Who can forget the Y2K hysteria in 1999, which arose from the fear that the speed of innovation in software development (specifically the adoption of IPv6, the new Internet addressing standard) would cause total collapse of our critical infrastructures?
Thus, instead of accepting the popular discourse that a crisis in software, and thus in software-driven-mobility, is indeed looming, I propose a more nuanced argument. Acceleration, slowdown and breakdown are all an inherent part of software driven mobility systems, and they always occur at the same time, mutually interacting with one another, creating a complex entanglement of varying temporalities. Doing so would also help us accept the inherent bugginess and lethargy with working with technical systems, and perhaps get out of constant hope that traps us into thinking that new technology will be able to save us from the problems of the older technology (Morozov 2013) – which it of course does not. Doing so helps us embrace endurance over newness, and maintenance and repair over quick innovation.

Conclusion

This paper showed how mobility systems, and likewise development of software for them, are intrinsically dynamic processes encompassing a plural landscape of temporalities - shaped by the interaction of sociality and technology in-action. This dynamism within software production results from a number of factors. Firstly, I underlined how software producers face a constant culture of acceleration of production and innovation of digital navigation software. Market competition within the software industry helps dictate this temporal order of how quickly software should be produced, which includes the customer’s product release timelines (like a new car being ‘launched’ onto the market), as well as software industry-wide competition with other mapping software companies. All this competition enforces a company culture that values constant innovation and ‘sprint’-based production. Software developers also learn about the bugs and breakdowns within their software in real-time, though analytics. This enforces an ‘urgency mode’ that their work has to be fixed and optimized, fostering ‘agility’ to change anything based on a customer or user’s demands.

Secondly, I highlighted that navigation software is also intrinsically run on optimizing the route’s efficiency, and finding the fastest route. Navigation is often designed with efficiency in mind – bringing people to their destinations in the fastest amount of time, using the least amount of fuel. Regional authorities now collect data about traffic, accidents, or special events that can be fed back into a navigation system in real-time in order to maintain this culture of efficiency and acceleration.

That being said, if we peek behind this culture of acceleration we can find a multiplicity of other forces – slowing down, stuttering, moving in reverse, or completely coming to a halt in breakdown. These multiplicities are linked to working with software – which, as I underlined, is a very complex, fickle material, dependent on human sociality and highly prone to breakdown. As we learned, these social factors include moments of negotiation between developers or their own personal temporal orders around their vacation time or their career timelines. Moreover, most software is not built from scratch, but built on years of legacy code that any developer has to deal with in order to make something new. The ‘legacy’ of a software system, the way in which various algorithms are optimized and run, or type or availability of data that is fed into the algorithm all feed into the dynamic and unpredictable nature of our mapping and navigation systems.

My ethnographic examples attempted to bring software development into the picture of how contemporary mobilities and mobility systems work. The popular sentiment around smart cities or our networked, seamless digital cultures makes it seem that these temporal characteristics of material infrastructure that limited us in the past can be reconfigured, that transport can be made more quick and seamless or capital can flow faster, as the immateriality of bits absolves us from the messy burden of our material world. The reality is quite the contrary.
Note

1. The temporal metaphor comes from Rugby, where a scrum is a sort of player-formation where players struggle to gain possession of the ball when it is tossed in among the team members (Cervone 2011). The term ‘sprint’ follows this sport-based metaphor.

Disclosure statement

No potential conflict of interest was reported by the author.

References


