Invisible Bits: Viewing Software as Infrastructure

Contrary to what might be expected, this thesis above all concerns itself with the boring, the extant, and the mundane, as compared to the novel, the disruptive, and the new. It presents an alternative to the frenzied pace of innovation and the seeming obsession with novelty which has captured society today. Within this thesis, I aim to reframe innovation's closest partner – software – in a way that favors continuity over invention. Software -- that amorphous term often taken to mean everything concerning computers aside from their physical hardware -- apparently threatens disruption from all angles, expanding into all areas of our lives. Yet, I argue that this innovation-focused viewpoint insufficiently captures the stability and complexity of software. This focus on the new coincides more widely with a neglect of the old, eschewing crumbling bridges for artificial intelligence. In response, I seek to contextualize and illuminate the too-often overlooked ways that software works invisibly in our daily lives.

I approach through the field of infrastructure studies, a burgeoning area of study which aims to better understand and unpack the many complicated technologies we interact with on a daily basis – all-encompassing, capitalized terms such as the Internet, the Web, or the Cloud – by framing them as infrastructures. Scholars in the field define infrastructures as taken-for-granted, largely-invisible networks or webs of interconnected systems. As one infrastructure scholar notes, perhaps it's best to define infrastructures negatively, as those webs or layers of systems "without which contemporary societies cannot function." It is the work of infrastructure scholars, therefore, to uncover and examine the social relations and technical characteristics of infrastructures and their components. Susan Leigh Star, one of the pioneers of infrastructure studies, refers to the field as a call to study boring things. Infrastructure studies employs an

inversion of sorts, illuminating and examining the boring and mundane rather than the new and novel.

With regards to my thesis, I employ two related approaches in infrastructure studies to my reframing of software. The first approach, which focuses on the overlooked historical and social relations inherent in infrastructures, inspires my second chapter on the history of software. Rather than seeing infrastructures as isolated, singular things, this relational approach sees them as systems and webs of systems dependent largely on context. In my history of software, therefore, I detail its many coexisting forms and conceptions, eschewing the idea that software has ever existed as a single form such as a shrink-wrapped package or application. The history of software actually suggests that software at all times forms interconnected systems with disparate dependencies and relations. In addition, the continuity of software becomes more apparent, with software implemented in systems such as airline booking since its very introduction as a concept in the 1960s and 1970s. Above all, taking a historical view of software diminishes its role as a novel, disruptive thing.

Having examined the historical aspects of software, the second infrastructural approach which inspires my third chapter aims to illuminate the technical and material bases of infrastructures as a way to understand their social contexts. This materialist approach includes, for example, looking at the technical mechanisms of video distribution on the Internet to better understand the organization and design of the Internet itself. In the case of software, I examine the technical details behind the interconnection of disparate software systems and services across the web. At all times, software has sought increasing modularity, and these sorts of interfaces allow just that. I look at APIs or application programming interfaces which connect software services through interfaces according to a uniform protocol. I perform case studies of two

particular API-based software services and examine how their services exemplify the infrastructural characteristics of software. Stripe has developed an API-based service which interfaces with payments infrastructure, allowing software developers to integrate credit card payments within any of their services. Twilio, in a similar vein, provides a standardized interface which allows software to integrate telephony services such as text messaging. Both these cases demonstrate the ways that software serves to expand and standardize existing infrastructures and systems rather than necessarily creating something new. Software continually must interact with legacy systems and their residual impacts.

I conclude by expanding upon what it means for software to be an infrastructure. Software does not exist as a monolithic whole or as a single form at certain points in time, but as many coexisting forms across decades. To capture the vast array of software's modular parts and coexisting forms in addition to its social and relational dynamics and tensions, I argue that we must imagine an infrastructure. Silently embedded in and layered on top of existing infrastructures, software is inseparable from their disparate functions. In our own work and play, the many layers of software have become taken-for-granted much like the lights overhead and roads beneath. While software at times seems to come to the fore—in viral mobile applications or websites—these applications represent not software but "experiences," detached from the layers of infrastructure enabling them. Software, then, is boring, mundane, unremarkable. While at once invisible, it has very real consequences. Its development is uncertain and seemingly untenable, wrestling with the legacies of old systems and visions of the new. Most importantly, software itself breaks down, ages, and decays. It must be maintained, updated, and repaired. In seeing software as an infrastructure, we acknowledge it as part of the "furniture of our lives. " In short, software turns out to be "nothing special."

Focusing on the old and existing necessarily draws our attention to the centrality of work that goes into maintaining these infrastructures. An infrastructural view of software suggests that we view software developers largely as maintainers who standardize and repair existing systems. As soon as a novel software service is released, it begins to break down and requires maintenance. This focus on maintenance therefore goes against the sort of "technical heroism" we see with genius hackers and developers. Software maintenance and repair deserve equal respect as "hacking" and creating. Additionally, the role of software developers themselves must not be overemphasized. Behind software services, a vast array of non-code-related labor serves to define standards, secure data centers, and tune algorithms. When we pay attention to the underlying labor, our experiences with technological systems change.

My aim with this thesis has been to illuminate to some extent the overlooked ways that software functions and performs. The next time you open up Snapchat or Instagram, I hope that you will consider at least briefly the diverse range of software powering your experience and to ask "where is the labor?" Above all, I hope that this thesis has demonstrated the importance of slowness and continuity in understanding technologies – the importance of not beginnings, but middles and ends.