Software as Composites: A Maze of Twisty Passages

Daniel "Drex" Drexler

Center for Science, Technology and Society at Drexel University

Abstract

A study of software, the way it materializes perspectives, and the limits of the promulgations of those perspectives.

Software as Composites: A Maze of Twisty Passages

Contents

It's Software's World, We Just Live in it

[It] matters with which ways of living and dying we cast our lot

D. J. Haraway (2016, p. 55)

Software is now well represented in every nook and cranny of the world. Though this project directly engages software, software was also used at every stage of its production: conception, composition, distribution and (in the age of covid-19) discussion. In the same moment that our reliance on software reaches ever loftier heights, we are surrounded by stories of important software projects being biased against vulnerable groups. Standpoint theory suggests that such biases often come from the hidden and unacknowledged perspectives of the people making software (D. Haraway, 1988; Harding, 1992). In this sense, the problems caused by bias in

software are not new problems. But, as Mackenzie (2006) and Kitchin and Dodge (2011) have noted, the management of a software project is filled with the work of managing relationships. Relationships between particular software artifacts (both the current version and older releases), the humans working on the project and expectations about the future. Software also exists within ecosystems: code written for a particular software project is surrounded and supported by code that many other software projects rely upon. How a particular software artifact impacts the world reflects the intent of both the organization that released this software artifact and the diffuse intent of the surrounding software/hardware ecosystem. Software interventions into existing projects can highlight the paths of agency within software artifacts. What can we learn about effecting change in software by studying how agency is mediated by the assembly process?

Software artifacts are the result of an assembly process that brings together the source code from many different software projects. Each software artifact is what Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999) called a "blackbox" - a container that hides complexity. But in order to properly describe software, I needed to expand the idea of a blackbox. In Latour's theory, each blackbox contains other blackboxes with clean separations between those layers (Latour & Centre de Sociologie de L'Innovation Bruno Latour, 1999, p. 183). Software blackboxes, however, often interact across blackbox boundaries. A software artifact will use one library as a clean blackbox but interact directly with the inner-workings of another library. Exactly which pieces of code drive particular behaviors in a given software artifact is difficult to predict in advance and working backwards from artifact behavior to code identification is not straightforward. This be because a software artifact is a composite - a heterogeneous mix of elements where the role of each element in any given software artifact is determined by the rules of the particular assembly process used to create it and whose component parts cannot be broken apart. This means that users and developers experience things in fundamentally different ways. Where users are always interacting with composites, whose various component pieces cannot be

differentiated within the composite, developers are always interacting with pre-assembly components. This is because changing composites directly is inefficient, so modern software development process relies on making changes to components before one or more assembly process(es). These components are where engineers leave imprints of their views. However, understanding how those views coe to the surface of composites depends on examining the assembly process that creates each given composite.The path of each materialized perspective through various assembly processes is unique.

Returning to the question of what one can do to address perspectives in software, this project takes the Free Open Source Software (FOSS) project Emergency Development ENvironment (Eden) as its object of study and site of intervention. I chose Eden because it is, as far as I can tell, a well-made piece of software that fulfills its role in the world. Others will engage socially problematic software and try to fix it, but the goal of my work here is to engage with a piece of software which is not obviously harmed by the standpoints of its developers. This does not aim to be a critical engagement, but an engagement whose successes and failures may be informative for future critical projects. The result of the modifications to Eden can be seen at <url>. The changes to Eden highlight where Eden crosses blackbox boundaries, though my efficacy in making those crossings apparent is limited to particular *kinds* of boundaries. As we will discuss, software is deeply heterogeneous and that heterogeneity impacts the work that is needed to engage with it.

Software: a Thing and a Process

> *I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer[.]"*
>
> *Ullman (2012, p. 3)*

It is difficult to talk with precision about what software 'is.' Individual programs that have been written in (or translated into) machine code are software (I will be calling these 'software artifacts'). Networks of software artifacts which only function

when connected (such as the combination of one or more Facebook apps and the Facebook servers) also are clearly software. There are also software projects like the IETF Task Force that produce no software artifacts at all, but create and manage standards that are essential for the interoperability of software. The field is diverse and engaging with it demands some definitional boundaries.

Mackenzie (2006) focuses on understanding how 'software projects' function. The software artifacts produced by (or associated with) a given project, the people who work for the organization that owns the project, and imaginaries about the future or the purpose of the project are all part of Mackenzie's definition. A software project is made of individual artifacts (both software and others) and relationships. This means including both the particular qualities of each software artifact (how does the current version of a program work and how the next one *will* work) as well as the relationships between each software artifact (how the current version will give way [or not] to the next version) in software. It also means that software is the thinking, planning, and imagining about how the project will: work, relate to the world, and relate to other software artifacts and projects (Mackenzie, 2006). Software is both material and socially defined.

The qualities that exist within a particular software artifact, were first imagined by engineers (much like Fleck (2012)'s thought collective) that work on that software artifact's software project or one of the supporting projects. The pipeline between how we imagine software will be in the future and the work of creating software that behaves like we imagine it should is important. It means that examining the details of how software works will tell us about how people imagined it would work.

The Material Character of Software

The material and social characteristics of software are co-constitutive its impact on the world. As Facebook attempted to respond the changing social situation around gender, they faced a back-and-forth between what was technically possible and generally allowed. They eventually settled on a solution that allowed lacuna to exist in the

database in return for a more socially acceptable list of gender options (Bivens, 2017). Softwares flexibility means that nearly any situation can be represented and recorded, but each new piece of software is built on top of an ecosystem which makes certain things easy and other things hard. The database practices Facebook relied on in their early days represented gender as a binary value - a choice which later made it impossible to both offer non-binary genders and have a value in the gender field of the database.

Sociological work on new media supports the idea that there is something unique about how software impacts the world. Many of the ways we socialize through software have pre-digital antecedents, but the unique power of software to create new experiences is clearly visible. You do not have to be making software to feel the power *of* software to structure and channel your life. We are both using media in was that are only possible through digital systems and using digital media to account for ourselves in new ways and to new audiences (Humphreys, 2018; Jurgenson, 2019). Though we have always been different things to different people, it is only through software that we can be interacting with people who treat us as if we are famous and as if we are totally unknown simultaneously (Dean, 2010). The important thing to understand is that these new social realities reflect the new material force of software being brought to bear on society. These new bottles contain old wine, but it turns out that the bottle makes a huge difference to how we experience the wine. It isn't necessary for software to be attempting to transform the structure of society, its material force drives social transformation even when that transformation isn't imagined by its creators.

Software Cultures are Cultures of Materiality

Ethnographic work on communities that are deeply involved with software support the idea that the material qualities of software uniquely and meaningfully impact those who work on it. When Kelty (2008) argues that the open culture of digital spaces follows from the commitment to sharing, he is also suggesting a strong linkage between the material character of technology and the social goals of its builders. It would be possible to build software *wrong* and undermine the Free Open Source Software (FOSS)

movements' goals (Kelty, 2008). His work underlines the importance of understanding the material qualities of software while it largely leaves them untouched.

The history of computers and computer work speaks to the material demands of software and the way that those material characteristics drive social factors. Computer workers gained their reputation for keeping odd hours and existing outside or alongside the traditional power hierarchy because they needed to do work on mainframe computers when those computers weren't doing business calculations (Ensmenger, 2012). The particular materiality of computers at that time (that they were singular and could only run one task at a time) gave rise to social characteristics have have somewhat endured to today[1]. When Cox and McLean (2013) investigated the connections between speech and code, the material force of software was central to the connection. One reason code is special is that it is speech that can be mechanistically executed as well as being the content of a traditional speech-act.

There are even pre-computer arguments for seeing computer software as having unusual and deeply important material realities. Hacking is a cultural movement that began before software and which is focused on joyful play in eliding and subverting the intended purpose of objects (D. Drexler, 2019; Gabriella Coleman, 2012). That hacking has become so associated with digital technology testifies both to the power of the act and the complexity of the task. Hackers enjoy hacking on software because the medium is difficult and demanding. They can make a career out if it because of the enormous material impact of changing software from working *this way* to that way.

Materialized Perspectives

A model is worked, and it does work

D. J. Haraway (2016, p. 63)

[1] I see a straight line between the historical social oddities of computer workers and the formation of the modern FOSS movement. Having a space where computer people already understood themselves as being "special" (even if it was special in an pejorative way) would have also helped them imagine that different property systems were possible. The early history of Linux was fueled by the unusual legal position of AT&T's Bell Labs (they were banned from making money off their work) which led to them taking few steps to protect their copyright and helping seed the open source movement (Kelty, 2008).

My account of the pluripotency of software and its particular material impacts begs a question of the nature of software: how much of its impact is intended and how much of its impact is caused by bias within the software? Software is basically epistemic - it functions through selecting and acting on understandings of the world. Understanding bias and developing language around bias is a well-studied area in its own right. Bias can persist in knowledge and practice in a peripheral way (Fleck (2012) describes how syphilis retains a moral dimension even when it is understood to be a microbe) or in a way that determines the entire structure of thought on a subject (such as Said (1979)'s deconstruction of orientalism).

The theoretical framework I find most useful here is Harding (1992) and D. Haraway (1988)'s "standpoint theory." Specifically I'm interested in the connection (or lack of connection) between choosing one way of knowing or doing within software over another way of knowing or doing. When one does this without acknowledging the choice or connecting it to a wider world-view, it's what D. Haraway (1988) calls a god trick: the appearance of knowledge simply *being*, as if it came from nowhere and is not touched by the life background of the person who created it. God tricks are central to systems which embed unacknowledged standpoints within them. These systems can begin problematic, like how early biometric techniques tend to split populations into groups because they were built with a eugenic frame of reference Subramaniam (2014). They can also accidentally become problematic, like how the fetishistic focus on the gene led us to over-invest in technologies that could not, on their own, provide us with new ways to affect the network of genetic and biological action that we call life Harraway (1997); Reardon (2017). Problematic characters of systems are allowed to stay hidden because the problematic qualities are not problematic for those who created the systems.

It is not necessary for a system to become problematic before it's useful to investigate standpoints encoded into it. Software, like any other tool, exists to help save human effort. In relieving humans from action, tools must trigger outcomes that might otherwise be proceeded by human choice. The number and nature of choices that can

be elided within tools have only grown more diverse with time. The real world work of finding and addressing standpoints in problematic software can be assisted by finding and addressing standpoints that are not themselves problematic. When Traweek (2009) studied high energy physicists, her observation that the particle detectors built by different teams had characteristics that flowed from those teams' personalities and preferences did not need to be used to repair a flow to be useful. This project is interested in finding choices that software makes but does not acknowledge. These latter day god tricks escape from documentation or acknowledgment. They can be deeply aligned with their creators' world view and be difficult to describe to insiders. A great deal could be learned by engaging with the engineers who create these materialized standpoints, but we don't need to speak to them in order to point out what is there. ,

Software Without Ethnography

> *[T]he "writing of technology" is by by no means universal; the opaque and stubborn places do not lie simply beneath technology, but are wrapped around and in it*
>
> *Mackenzie (2006, p. 181)*

The bulk of sociological work around software has been work that with the material effect of software through its impact on social worlds. This project does not do that. It is conducted with the assumption that omitting social engagement will limit, rather than eliminating, the usefulness of the project. This belief is based on the studies of social outcomes that suggest that software possess a strong material character outside of social construction (Dean, 2010; Eubanks, 2018; Humphreys, 2018; Kelty, 2008). It is also based on work that show that once social information has passed into digital systems, that information will be manipulated and used in ways that can be driven as much by the quirks of digital systems as by social goals (Cheney-Lippold, 2018). This project, I believe, points towards a future project that engages more deeply in the social worlds of the makers and users of Eden. Doing so would allow us to

understand how Eden constructs space for its makers, its users and the world in general (Kitchin & Dodge, 2011).

Better understanding the nature of software isolated from social situations will enable more complex work that studies both. It is those qualities of the digital world, what is found when you "pass through a membrane where the real world" no longer matters, that this project is interested in finding (Ullman, 2012, p. 3).

Yes, but what IS Software?

Work on the powerful impact of software on our social world (and the power impact of our social world on software) somewhat elides the central question of my work: what *is* software? What are the qualities of this thing and this practice that has transformed our world, often on accident? Cramer (2008) focused on the Janus-like qualities of computer languages - forced to fulfill the needs of humans and machines. While Cramer (2008) notes that the only actual computer language are the machine languages spoken by computer chips, he also points out that the primary concern of general purpose 'computer'[2] languages (including python) is attending to different human priorities. This is especially true because all general purpose languages can be reduced to one another, so any difference in functionality come only from the difference in how well the language suits the preferences of the human writing it. Yuill (2008) talks about the character of the ecosystem of programs and the centrality of "interrupts" to modern system design. Most software written today is written with the expectation that it may be interrupted. This is because modern computers remain responsive to people (and other software artifacts running on the computer) by being ready to interrupt any process at any moment. This is important because it prepares software to try to deal with the unexpected.

---

[2] 'General purpose' programming languages are all Turing complete, which means they can fully describe the behavior of a 'Turing machine' - a theoretical automaton described by Alan Turing in the early 20th century. This means that all general purpose programming languages can be reduced to one another (a series of instructions generated by one language can be generated by another). However, there are also 'domain specific languages' that are designed to solve more specific problems that cannot be reduced to one another (and cannot fully describe the behavior of a Turing machine). Ironically, this document was typeset in LaTeX, which is Turing complete.

I find Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s blackboxes a good start[3] for talking about software and how it functions in the world. Blackboxes are enclosed units that accomplish a task without exposing outsiders to the interior details of how it works. Like software, blackboxes can contain almost anything with any level of complexity, but to those outside the box it appears singular and cohesive. It's only in failure that blackboxes are differentiated from other objects. Latour was interested in how, once a blackbox failed, the fact that it has components becomes socially available and socially unavoidable. Often, blackboxes contain other black boxes, each one waiting for failure to make the fact that it has internal components. Software too, has important internal components that are invisible. However, because the tools and skills needed to disassemble software are so much rarer than the skills and tools to disassemble, in Latour's example, a projector there are important differences in how software is perceived socially.

Taking Action

> *We live in capitalism, its power seems inescapable – but then, so did the divine right of kings. Any human power can be resisted and changed by human beings.*
>
> *Le Guin (2014)*

So in one hand I hold Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s blackboxes and in the other Harding (1992) and D. Haraway (1988)'s situated knowledges. As Mackenzie (2006) recounts, the process that produces software is complex and heterogeneous. It defies simple analysis and it, itself, a deeply social thing that will also produce software artifacts that can be mechanistically executed (Cox & McLean, 2013). Yet we are beset on all sides with software whose bias is plain for all the world to see (Bivens, 2017; Dean, 2010; Elish, 2019; Eubanks, 2018; Schüll, 2012). How can our academic theories be used to address these practical concerns? How

[3] Computer science draws on the general blackbox concept that Latour drew inspiration from to talk about acting on formal function definitions without knowing about how the function is implemented (Abelson & Sussman, 1996, p. 33).

can we find what Sismondo (2008) called an "engaged program," which combines theory and contact with the real world?

The first step to to try doing real work on real software. Zuiderent-Jerak (2015)'s *Situated Intervention* (which also draws from situated knowledges) argues forcefully that social scientists can and must intervene to learn. He describes how medical professionals use interventions (and their outcomes) as tools to generate knowledge and test assumptions. My own pre-academic experience as a software engineer supports the knowledge generating power of careful and measured intervention. This project hopes that it can find materialized perspectives within a wholly unobjectionable software project (and its associated software artifacts) and, though engaging with the source code of that software project, learn something about how we might address materialized bias in more convoluted situations.

The object of study

Software in General

Before diving in, I should define some language. Some of this language will be drawn from existing literature and some of it will simply be me drawing boundaries around the sometimes fuzzy concepts that can be involved with software. I will be developing these ideas throughout the paper and some of them exist in an updated form in the glossary.

A **software artifact** is any arrangement of machine code, scripts, configuration files, network connections, or other constitutive resources that can be executed by a computer. A collection of such resources become a software artifact as soon as the computer can be passed the bundle of elements and follow the instructions encoded within the artifact. This form of software is important because it is in this form that software is distributed.

A **software project** is, as Mackenzie (2006) defines it, the extended social, material and organizational structure that produces "software." These projects are not unified by the production of software per-se, but by hoping to change how software

functions or how it impacts the world (in a Kitchin and Dodge (2011) sort of way). Though software projects may produce software artifacts, they do not have to. Technical organizations that create standards that other software projects might follow are also software projects. The same is true for professional organizations that aim to shape the culture of software production.

Throughout this paper I will speak about the intent or actions of a software project. This is not an attempt to say that the software project totally controls the actions of individuals involved with it, but a recognition that the social conditions inside a software project are expressed through individual members of that software project taking action. Each individual person has their own explanation for why they acted in their particular way, but we can still usefully speak about the general opinion, actions and intents of the general project.

Finally, an **assembly process** is the arranging of elements of a software artifact in such a way that the computer can execute them. Such a process always happens (even if it is just placing machine language instructions sequentially in memory) before the computer can execute the software artifact. This process is often carried out by one or more software artifact(s). Sometimes these software artifacts are part of the software project making the software artifact, sometimes the are tools shared between many software projects (generally with configuration supplied by this particular project).

How I Engaged with Software

What is EDEN?

> *That virtual worlds are places means they can be fieldsites;*
>
> *Boellstorff (2015, p. 107)*

The particular piece of software this project studies is Emergency Development ENvironment. Eden was originally created by a coalition of Sri Lankan Information and Communications Technology companies in the wake of the 2004 Indian Ocean

Earthquake and Tsunami. The project is now managed by the nonprofit Sahana Foundation. It has been used in numerous disaster responses since 2004 and is also used by a number of organizations for managing resources outside of any specific disaster (Sahana Foundation, n.d.-a). Eden's functionality can fairly be described as tracking and managing resources. Its power comes from the enormous range and detail of the information it knows how to track. Eden has indexes for organizations, people, projects, events, facilities, supplies, documents, possible scenarios, and events. It also provides on-platform messaging and special tools for the management of emergency shelters (Sahana Foundation, 2011). Each resource is meant to be connected to other resources: people work for organizations, projects are run by organizations and are associated with people, facilities are linked with projects and organizations and other resources like supplies and verticals. The flexibility of the software means that it can be used to manage a single organization, multiple organization, or used as a hub for many different groups coordinating around shared goals but without central authority. The system aims to be as comprehensive as possible and to use the same entity for resource and system management. The entries for people can act as a simple rolodex and will also double, if desired, as that person's account within Eden. The interface for managing goods is the same interface for managing access to the management system. To the degree possible, Eden is structured to allow the same people who do the day-to-day work of the organization to administer the system that manages the organizations' resources. It also aims to be an effective tool for people at all levels: administrators can track where people and supplies are allocated and volunteers can access their assignments, documents and information about how to use resources.

Why EDEN

I selected Eden for this project for two primary reasons: it is a Free Open Source Software software project in a language I am familiar with and it is not obviously problematic. I will detail its open source pedigree shortly, but first I want to explain my second selection criteria. "Improving" software (whatever improving means) can usefully

be separated into at least two questions: what quality should change and how do you bring that change about? This project attempts only to speak about the second (likely easier) question. Partially this is because the project was designed to fit within the confines of a final project for a masters degree. This project represents, roughly, three graduate level classes worth of work. I didn't feel like I had time to find *and* fix flaws. So this project should be understood as neutral on the question of, "is Eden good?" Nothing about Eden seems bad to me. They have received numerous awards, glowing testimonials, and are used by many large organizations that could use other products if Eden were lacking.

I lack the experience and expertise to say that I think Eden is *good*. Assessing if software is "good" or "bad" is not straightforward. Simply examining software artifacts in isolation tell us very little. It's only through engaging with one or more software artifact(s) as they exist in the lived world and contextualizing that with detailed ethnographic work that it's possible to start making value judgments (Eubanks, 2018; Schüll, 2012). Those judgments wouldn't be universal, of course, but would be about a particular population. So this paper is done from the perspective that Eden seems fine to me. The project is interested in the technical details of how the Eden software artifact emerges out of its software project and how agency is modified by that process.

Finally, and briefly, this project does not take the position that the technical qualities it investigates are free from the impact of social structures. Gabriella Coleman (2012) and Kelty (2008) have both compellingly shown that the social and technical co-produce each other. However, we can still usefully speak about qualities that technical systems have and how those qualities impact our lived experience. That this project has obvious extensions in the social and ethnographic realm is a strength and declining to investigate them should be understood as a concession to time.

Out of Many, One

Wherever possible, Eden uses FOSS technologies. The language it is written in, the libraries it relies on to provide functionality, the tools it uses to support its

functionality, and its main operating system are all both open source and available at no cost. Eden will generally operate on top of non-FOSS systems like Microsoft Windows, but the team doesn't prioritize systems outside of the FOSS systems they develop and test on (Sahana Foundation, 2015). Kelty (2008) talks about how FOSS is both a philosophy and a system of development that has practical impacts. One of the side effects of Eden committing to use the FOSS ecosystem is it makes my form of engagement possible. Though it is possible to examine compiled machine code and draw some conclusions about the intent and process that assembled it, such a project would be far outside my capabilities. Instead, the source code of Eden, web2py (the web framework Eden uses), Python (the language Eden and web2py are written in) and all of the libraries used by the project are open source. Their preferred databases (MySql or PostgreSQL) are also fully open source projects. Open source projects often don't just publish the current source code, but offer full histories of what change, when it changed, who changed it and how it was changed. These changes often include notes about why *that particular* change was made over any other possible change (Chacon & Straub, 2014, p. 13-16). Open source projects also commonly have systems for tracking lists of unfixed flaws as well as planned future improvements (both types of Mackenzie (2006) relationships), but those systems operate above the layer of source code so this project does not engage with them.

Enumerating The Specifics of my Engagement

Eden has a large number of individual components. Executing a fully assembled Eden software artifact would involve a number of Python libraries, a number of other software artifacts (a database, various compiled libraries) and a large amount of non-python code (such as the javascript contained in the Eden repository). Engaging all of this code is certainly possible if given enough time, but did not seem possible given the constraints of a masters project. So, I have chosen to focus on the Python portions of a limit set of the libraries used by Eden. These libraries are listed in Table 1[4].

[4] The exact versions of the components may not be the ones stored in the canonical git repository. Each repository was at a particular git hash for this project. Eden was at

Table 1
Purpose of libraries

Component Name	Brief Description of Functionality
Eden	Implements the logic for managing the various resources Eden manages
web2py	Provides interfaces to easily provide web services over HTTP and related protocols
pydal	Library for interacting with a database using Python objects instead of raw text
yatl	Library for generating HTML content through templates

My choice to focus on these components in particular does not come from a considered opinion that these four components are the most important components. That judgment would require a good deal more work than I have done to develop an opinion about how to compare functionality across programming language boundaries. For the same reason, my analysis omits all non-programmatic components (databases, static content, etc). This omission should also be understood as a limitation imposed by time as opposed to a considered opinion that those components contribute less to the behavior of a software artifact.

Instead, I chose these components because they were the ones I interacted with most during my investigation of the Eden project. There are also formal links between the projects: the Eden project identifies web2py as its Web Framework and web2py identifies both pdal (as DAL) and yatl components of itself (Di Pierro, 2020; Sahana Foundation, n.d.-b). There are other primary components of the Eden project, but many of them are in other languages or are themselves software artifacts (such as databases).

It's foreseeable that my conclusions, focused on these four components, might not extend to the entire software artifact / project, but I doubt it. In almost all circumstances, the components that most impacts what the software artifact does would likely be in the Eden source code. That does not mean, as I will show, that most

---

0718c0681bb58576a613e0edc4d4070ac214be21. web2py was at 93ef108c0b4b1c100622bf0002ec0972dec8be46. pydal was at cecd77127c122404c1aee7f6377c6a0150d86d84. yatl was at 5deb403a9e45f617588f02cd8f7682b3f98571b4. Some of these only exist in my fork of the respective repositories.

of the code that is associated with a particular element of an Eden software artifact is within the Eden source code. It may be useful to refer to the size (in lines of code) of the various libraries to get a rough sense of the relative size of each one (See 2).

Table 2
Size of Libraries

Component Name	Lines of Python Code	Lines of Non-python Code
Eden	469,117	793,661
web2py	103,593	41,793
pydal	24,518	416
yatl	1.131	69

In general, the number of lines of code in a piece of software will correspond to that programs capabilities. Lines of code can also be used to roughly compare the relative contributions of components, especially when all components are written in the same language. It's possible to write lines of code that have no meaningful impact and it's easy to imagine scenarios where single lines of code make thousands meaningless (by turning off a feature, for instance), but there is no reason to believe these instances would happen more or less often in any particular component. Similarly - it is much easier to imagine accomplishing the work done in fifteen lines of code with ten different lines, but it is much harder to imagine doing the work of fifteen hundred lines of code in a thousand. Efficient code is efficient, but there is a limit.

Traveling in EDEN

My work began, as all software work does, with work of assembling the components that transform Eden from a collection of python source files into a software artifact. This requires connecting a number of components together: libraries and tools in the Python language and a wider set of heterogeneous tools (databases, non-python libraries and tools). Some of these components remain as source code to be interpreted by python, some of them are downloaded as source code and go through their own assembly process to generate their own software artifacts. The assembly process that draws the components of Eden together does not happen in a single step, but is a number of loosely ordered heterogeneous processes. Some must be completed before

others and there is an officially recommended order in which to prepare the various components, but often these orders are more matters of preference than necessity. What is important is that all of the required components are collected in their expected places by the time Eden itself is run.

The Sahana Foundation recommends that developers use a Virtual Machine that has Eden and its supporting libraries installed through a script. However, at the time that I engaged with the project, the instructions for how to install Eden focused on using operating systems and python versions that have since stopped being updated[5]. Though more modern scripts can be found, the standard install scripts for the Eden Virtual Machine and the script to install the software on a physical machine both use operating systems released in the mid-2010s that have not been maintained for at least 5 years (Canonical, 2020). However, Eden is aware that this is problematic and makes an effort to support newer versions of python and newer operating systems, but official instructions to use the latest software are sparse.

Approaches to Tooling. Choosing to offer a fixed and unusually old set of tools is the first materialized I found encoded into Eden[6]. The state of the scripts reflect a culturally important choice. The team could note and situate their approach in

---

[5] As Mackenzie (2006) points out, software is a process not a thing. This is especially true for software like Eden, which is made available to users over network connections. Pieces of software that stop receiving updates can quickly become vulnerable to new interactions with an ever-changing ecosystem. These vulnerabilities can result in programs becoming unstable, new security vulnerabilities, or simply an inability to keep up with the changes to other components. Software that is no longer being "maintained" (the phrase used to describe engineering work to patch security holes, update interfaces to modern standards, and make those updates generally available) can suddenly stop working because one or elements of its ecosystem cuts off old software. It is true that isolated software can run for years or decades without changes and that, for some systems, users never experience adverse consequences for leaving them frozen in time. However, whether or not a program can "safely" be left alone is highly dependent on external factors and software that seems "safe" from needing changes can suddenly and tragically need emergency updates at high cost (Lee, 2020).

[6] It's a fair question to ask where Eden begins and ends. Certainly its source code is part of the program, but I would also include the collected documentation on how that source code should be deployed and supported as part of the "program," even thought it's not actively involved in the functioning of an assembled software artifact. This is best understood with Star and Griesemer (1989)'s idea of boundary objects - coordinating objects that are able to play multiple roles for different actors. The software artifact of Eden cannot exist for anyone outside of its immediate development team without distributing instructions on how to assemble the software artifact, so I include that material in the Eden boundary object. It's worth noting that many of the requests for help with Eden on the Sahana Foundation mailing list are questions not about Eden per-se, but about the tools and libraries that support a fully functioning Eden. In a very real sense, the Eden that exists without instructions on how to assemble it isn't available to me: I wouldn't know how to access it.

their documentation, but they have not. However, project-wise choices to use older or newer versions of tools are common. There are software projects (sometimes entire programming languages) that greatly value using the most up to date version of a tool or library (Fernandes da Costa, 2017). These communities have legitimate fear that older versions of tools are examined less often and will likely be fixed less frequently. They feel that using the latest (within reason) version is a goodness in of itself[7]. If the Eden team followed this philosophy, they would not default to using older operating systems and versions of python. They would need to watch their ecosystem of tools and libraries more closely, but the versions they were watching would also be more in the collective consciousness of the FOSS world.

All this isn't to say that the Eden team choice isn't a sensible one. The latest versions of tools often lose track of the world outside the top of the capitalist pyramid. The same networks of power that structure the spread of physical technology also shape the attention of active software engineers. Eden has chosen to use established and well-supported versions of their tools[8]. Their older set of tools is more likely to have undiscovered (or unpatched) security vulnerabilities, but they are virtually certain to function properly and immediately allow the successful assembly of an Eden software artifact. That set of probabilities seems very well suited to the expected audience for individuals first setting up their own copies of Eden: individual developers setting the system up to better understand it. When another team decides to deploy their own copy of Eden in a way that's available to the public, they can assemble that software artifact using a balance of newness and stability that feels comfortable to them[9].

---

[7] A language has evolved to speak about the relative stability of a particular software artifact (or, more precisely, the boundary object that includes a software artifact and its assembly instructions). Signaling about how often you will be changing your software allow users to choose the version of you work that will fit their own rhythm (Canonical, 2017). 'Unstable' releases are often more likely to have bugs or be insecure, but also have new features that cutting-edge projects may need. 'Stable' releases have few new features but are well understood and unlikely to be the source of problems.

[8] As of the writing of this paper, the versions of Ubuntu that Eden installs into by default have left long term support. Though evaluating if that's good or bad is outside the scope of this paper (and my expertise) it seems important to mention it.

[9] Of course, this update process also has risks. A well known mix of library versions will work together, but as I'll note later, mixing different versions of libraries can cause unexpected dysfunction. Suffice to say that the old versions of the tool and library ecosystem will generate the fewest unexpected tooling

Investigating Assembly

It's one thing to identify where the Eden install scripts sit in a cultural gradient. It's another to engage the particular choices that the Eden project has made. I chose to install a far newer set of tools, based on the projects newest recommendations for new installs (König, 2019a). I was guided by the python 2.7 installation script provided on the Sahana Eden wiki[10]. However, I chose to use a version of python originally released in 2016 (six years after the default version used by Eden). Using a newer version of python seemed like a sensible software engineering choice and also a way to check if Eden was, as they claim, able to operate equally well on the version I selected (3.6.9) and the 2.7 version that Eden uses by default[11].

Selecting a more modern python turned out to have knock-on effects. Some of the libraries that Eden requires are written only for python 2.7 and replacements were needed to function with 3.6. One of the major changes was that the version of web2py the Eden scripts installs does not support python 3.6 and so I needed to chose an updated version. I selected 2.18.5, but this new combination of the Eden source code and the web2py source code revealed that the two projects had drifted apart in small but essential ways. Specifically, in a section of the web2py and Eden code that handled data validation.

Data validation is, in general, the work of verifying that information conforms to a set of standards and informing the user how it has failed to conform when a problem is found. This should always be performed any time data is stored in a system. Digital storage systems have expectations that, when violated, can cause immediate systems failures. Web frameworks commonly provide facilities for verifying data and web2py is

---

problems and maximize for developing in an environment familiar to the Eden team.

[10] The Script I used can be found `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Linux/Server/CherokeePostgreSQL`, though as I describe, I often used different components. Unedited notes on the things I ran into can be found at `https://github.com/aeturnum/masters_project/blob/master/Notes/Web2Py_Notes`.

[11] I selected a release from the 3.6.X series because I'm fond of f-strings (a features added in 3.6 that allows compact formatting of printed information). However, 3.6.9 is now actually out of date, as 3.6.10 was released just a few days before the end of 2019 (Python Software Foundation, 2019). In a perfect world I would have used 3.6.10, but as this project includes no private information I stuck to 3.6.9.

no exception. Web2py has components that will check that that user input can be converted to numbers or dates or email addresses, among others. Web2py is open source and so all of that functionality can be expanded by programs that make use of web2py and Eden does so in a number of places.

Shifting Relationships

In between the version of web2py that Eden uses by default and the version of web2py that I selected, the control flow of data validation has changed slightly. This change meant that custom data validation classes that were written using the older web2py system no longer fit into the expected control flow of data validation. The Eden data validation code, based on the older standard, is not called properly when using the newer version of web2py. This doesn't stop the successful assembly of an Eden software artifact, but once that software artifact is assembled most of its functionality in inaccessible because the Eden-specific data validation code won't run and so the system won't accept that data.

This problem is interesting because it turns out it captures a moment in time between the web2py team and the teams that use web2py (i.e. Eden). If I had selected a slightly earlier version of web2py, then it would have used the old system and the Eden code would remain functional. web2py immediately realized the problem and integrated a fix[12] for the issue. This shows the management of relationships that Mackenzie (2006) describes. There is the work of making software and then the subsequent and parallel work of adjusting that software to stay within relationships boundaries. The web2py successfully implemented a new way of handling data validation control flow, but they neglected that they were doing it in such a way as to break an existing relationship they did not mean to break. Because I happened to select a version of web2py that was temporarily out of relation with Eden, I failed to assemble a fully functional software artifact. This was what got me interested in the assembly process itself.

---

[12] Interestingly enough, the problem was fixed by a piece of code written by a member of the Eden team (König, 2019b). The web2py team had already noticed the problem and thought they had fixed it, but their fix did not work for for all circumstances and required more work.

Investigating Assembly

Assembly is a label I'm using to cover a variety of software processes that source code (and many other things) and produce software artifacts. Almost all software is written in programming languages that have the dual goal of being possible for humans to understand and being suitable for mechanical translation into machine code. Machine code is, after all, the only language a computer can really understand (Cramer, 2008).

Methods of assembly vary widely, but they can roughly said to lie between compilation and interpretation. The purest compiled language will generate a software artifact that contains all the machine code that will be used by the software artifact into a single contiguous piece of machine code. That machine code will generally only work on a single hardware architecture, but the same source code can be assembled[13] into machine code on other architectures by different assembly tool-chains. On the other side of this spectrum are interpreted languages. A purely interpreted language assembly process does not involve the generation of any new machine code. Instead, it uses a standard software artifact (called an "interpreter") that reads the source code of the interpreted program in its human readable form like a recipe: reading the file(s) as the interpreted program executes and using machine code that exists within the artifact of the interpreter to tell the hardware what to do.

There are also a variety of methods for software artifacts to share access to centers of functionality. The most common is sending calls "out" to libraries. The pure compilation situation described above really only exists on hardware without a multitasking operating system. For every software artifact on your phone or computer, the software artifact will frequently make a call to a library (which itself sits somewhere between compiled and interpreted) where it specifies which library, which section of the library, and the information it would like to pass off. Then the operating system pauses the execution of the machine code in the software artifact and loads the library (or the

---

[13] The assembly process for a compiled language (and in fact for many languages that sit somewhere in the middle of the compiled-interpreted range) is generally executed by a software artifact called a "compiler." I'm avoiding using the common word to avoid confusion in with the range of approaches in assembly.

library's interpreter) to perform the requested task. When the task is done, the first software artifact is loaded again and told the result.

Most languages and software artifacts sit somewhere between being fully compiled and fully interpreted. Python is largely an interpreted language, but it also creates non-human readable bytecode versions of each python source file that allow the interpreter to execute more quickly (The Python Software Foundation, 2020). Python also allows access to libraries written in machine code and Eden invokes machine code libraries through python interfaces.

Digital Relationships. These libraries exist on the border of all of the software artifacts that use them[14]. They contribute to how the software artifacts that use them behave. Flaws in libraries can allow the programs that use them to be taken over. Sometimes these flaws only require attention to discover, but it's often the case that other elements of the hardware / software ecosystem must change before they can be found (Huang, 2003). All this is to say that the kind of assembly problem I encountered in assembling Eden is not particular to Eden. It's a universal concern when assembling software artifacts that use any functionality that resides outside their own source code (which nearly all do).

This system of linking together different code written by different people is an example of the relationships that Mackenzie (2006) found at the center of the software projects he studied. At the project level this means that a the people working on one project must pay attention to the development of many other projects. Are new versions of the libraries that you use being released? Will the old version be updated if security flaws are found? Does the new version have features or changes that would be useful for our project?

What is striking about the problems encountered while trying to assemble my own artifact of Eden is that, while every individual who assembles an Eden software artifact

---

[14] Libraries do not have to exist outside of machine code. When a library is 'statically-linked' to a software artifact, the section of the library's machine code that is used by the software artifact is copied into the software artifact's machine code. So when the statically-linked assembled software artifact calls the statically-linked library, the computer simply loads the section of the library that was copied into the software artifact code.

will need to go through their own process, the problems I encountered are particular to using an un-patched version of web2py 2.18.5. The install script I used contains other fixes for other aspects of that particular pairing of components, as well as different fixes for different combinations of components (Sahana Foundation, 2018). Eden can be assembled using an extremely wide set of components. It can run on that haven't been considered up to date for over ten years ago and components that are still considered cutting edge now (as well as many mixes between these two extremes). The Eden software artifact is designed with this capability in mind. It is a strength of the FOSS ecosystem that such a wide range of compositions is possible, but it also means that Eden can be a nearly limitless number of different software artifacts whose qualities emerge both from Eden's source code and the interactions between the particular versions of the components.

The Social and Material Components of Software. Keeping in mind the range of possible component sets for Eden, think back to the particular set used in the install script. I speculated the set in that script were selected because they're well known familiar to the Eden development team. They minimize the chance that people will mistake a problem caused by assembly for a problem that exists within the Eden source code.

When a particular software artifact fails to function as expected, there are many possibilities as to why. It might be mis-configured, the resources it needs to function properly might not be available, there may be an internal incompatibility between components, or there may be an error in the Eden source code. There are many functional Eden software artifacts that contain component combinations that the developers have never tried and have no experience with. When what appears to be an error is reported to the Eden team, often the first questions they ask are about the versions of the libraries that were involved. If the versions of the libraries being used aren't considered to be the correct ones, the bug can be closed as wholly or partially invalid (trendspotter, König, & Boon, 2020).

This method of engagement around questions of proper assembly and proper

component selection are, above all, practical. Over time the number of possible combinations of all the versions of all the libraries will inevitably expand beyond the abilities of any team. To control this somewhat, teams will have defined periods of support and lists of appropriate versions of components and only fix problems that found within that somewhat limited domain (Canonical, 2020; König, 2019a). But software artifacts that are assembled using components drawn from outside the official set of components may still function. Being 'out of support' does not mean that things will not work. It means that the team managing the software project has no opinion about the relationships between their code and the components you're using. They will support the official supported combinations and leave you to do your best with the unofficial ones.

This suggests a limit on the boundaries of the Eden software project. Maybe the unofficial software artifacts that the Eden team won't support now aren't really Eden software artifacts. However, I don't think it's that simple. What components (and versions of the Eden source code itself) are inside the 'official' Eden change over time. What is an unofficial Eden software artifact might one day become an official software version as the Eden software project changes its thinking about relationships. By the same stroke, official versions will often become unofficial. The team recently announced that, with the end of official support for python 2, they will be moving to using python 3 for all of their builds and updating their install scripts to use python 3. Suddenly, my Eden software artifact, which already uses python 3 is within the official definition and the Eden software artifact assembled by the script on the Sahana website is unofficial (König, 2019a; Sahana Foundation, 2015).

Reckoning with Failures

After assembling a functional software artifact I began to explore how my Eden software artifact functions on its own terms. I did this by entering the high level details of this project into Eden's project tracking feature. This included creating an entry for Drexel University's Center for Science, Technology and Society, an entry for my project

within that organization, and an entry for myself as the manager of that project. During this process I encountered both a bug in the source code of Eden and another unacknowledged materialized perspective.

The form that Eden uses to record the details of an individual has a field for a phone number. There's no indication that this number number should be formatted in one particular way over another. I didn't think anything the lack of instruction until my Eden software artifact informed me that I needed to "Enter a valid phone number" (König & Drexler, 2020). I didn't understand because my phone number was following the common conventions for phone number formats (I tried the 'xxx.xxx.xxxx' as well as 'xxx-xxx-xxxx' and 'xxxxxxxxxx'). Upon review of the source code of the file responsible for the message I discovered two things: that, based on a setting I was not aware of, the form was expected a phone number in international format (beginning with a '+' and a country code) and that the code was returning the wrong message (complaining that the input was not a phone number instead of complaining it was not an international phone number). After entering my number in the international format, my profile was created.

This felt like an excellent situation to investigate. It appears, on first brush, that there are two layers of unacknowledged materialized perspectives: the first being requiring numbers to be in the international format without informing the user filling out the form, the second appears to be an error in the message returned. My background as a programmer and my access to the source code makes me want to say that the Eden project intends to require international numbers and failed to return the correct message about that requirement. However, I think it is worth considering how these two problems appear without access to the source code.

First - imagine if the second problem was fixed (the software artifact returned the proper error). When the user enters a number without a country code, the Eden software artifact will reject their number and tell them it must be international. They now know what to do, but had no way to learn about the requirement before they failed to meet it.

Second - imagine if the software artifact informed the user they needed to enter their number with an international country code. But, when the code was left off, told the user their number wasn't valid. Again, the user would not be sure what to do to advance. It might be easier to find out (because the software artifact has labeled the field with the requirement), but *from the users' point of view* these two situations are very similar. It is only the ability to look into the source code that clearly sets them apart.

So, after discovering it, I decided to see what I could learn by giving back to the Eden project and fixing this (extremely minor) bug. I wrote and submitted a small patch that intended to fix the problem and check that the proper error message was returned. Unfortunately, much to my embarrassment, my fix was written incorrectly - it incorrectly eliminated another error message that could, in certain circumstances, be displayed to the user. As a result my fix was rejected, though once the Eden project was aware of the bug they immediately wrote their own fix (König, 2020).

In the discussion about why my fix was flawed, one of the Eden project developers said several interesting things about how he (and by extension the Eden project) think about software.

> "And no, there's no test case as to what error message exactly is returned.
>
> Whilst we could unit-test it routinely, **regression-testing too much detail can add excessive rigor thus make development harder**. **A wrong error message is an annoyance, but neither does it break the intended functionality**, nor does it jeopardize the data integrity - so it is low priority and should probably not be CI-tested." (König & Drexler, 2020)[15]

There are a couple of interesting ideas in this quote that, I think, can help us understand how software (or at least the Eden project) functions. First, there's the idea

[15] Emphasis added by me. This quote mentions "regression-testing," which is worth briefly explaining. Regressions are when old, previously fixed problems reappear in new versions of the source code. This style of testing is in contrast to "test driven development," where tests are written as a description of what code should do (Agile Alliance, 2015).

that testing too much makes development harder. Since König is talking about regression testing (testing of fixed bugs) the difficulty would likely come from tests unexpectedly failing after changing related code or would come from work needed to update the test when changing the functionality it is testing[16]. In any case, the decision to limit testing is also a decision to limit their automated testing to a certain subset of all program functionality. That development would be harder if the full scope of possible functionality was tested and that it is better to be ignorant about ancillary behavior like error messages.

The second idea is that accepting or rejecting the number based on its use of an international country code (which it does successfully) and informing the user of what they need to do should be considered separately. That however the program appears to to the user, if it accepts numbers in the desired format and rejects others then the component is working. The statement reflects a world view that, if a component is expected to have multiple outputs for a given input, each output should be considered working or non-working separately and that there is a hierarchy to which functions are more or less important.

Taking Action

A primary motivation for this project was the sense that situated knowledges felt like an unfinished tool for the practice it was critiquing. Our ability to see the truth is impacted by our life circumstances. Individuals cannot identify false truths that only appear true because that individual exists within a structure of power (D. Haraway, 1988). Situated knowledges explains the mechanism through which we inappropriately consider things true or objective, but it is of little use in understanding to affect changes to the things impacted by our limitations.

If software projects incorrectly believe things because of the collective

[16] It's here where I really miss an ethnographic component. A lot of my logic here is necessarily consequentialist based on my understanding of how software development works, but the logic the team is following could easily be deeper and more complicated than I imagine. Nevertheless I hope to draw conclusions that will be based on the material outcomes of their decisions and don't depend on the Eden project possessing a particular internal understanding.

backgrounds of their members, convincing them of that truth does not help them address it. The same cultural limitations that hem in action would also seem like they would hem in the process of selecting outsiders to help address the problem.

So I hoped to find ways to understand better how to address the standpoints of development teams through any materialized perspectives I might find within a software artifact. I took a wide view of what "address" might mean - it could be a direct improvement, it could be a way of highlighting alternatives, it could be tools that help teams see the boundaries of their own perspective. As I worked, however, it became clear that without an ethnographic engagement this goal faced real difficulties.

Improved for Who

No sociological work is needed to learn how to improve the Eden software artifact from the point of work of the project. If I want to improve Eden, doing so is simple. The Eden project Github has a number of minor issues with details about what change is needed to improve the software artifact. They also communicate about their long term plans and are open and encouraging of outsiders and users alike joining in to comment on what they would like to see in future Eden software artifacts. These conversations share the qualities that make accessing them difficult to users who aren't members of the right social groups (a focus on technical details, cultural signals of developers), but those are problems shared by the world of software and don't derive from specific qualities of the Eden project (D. d. Drexler, 2019; Kelty, 2008).

My decision not to engage with populations that use Eden software artifact means that I have no reason to believe I can understand the interests or needs of people who might use the software. Even taking Kitchin and Dodge (2011) into account and proceeding from the point of view that the software might impact space in particular ways outside of particular individuals is not much help. I am not experiencing a disaster[17]and even if I was it's unlikely that I have a sense of how Eden, among all the factors, was impacting me. That kind of work benefits from longitudinal work that I

was not planning or prepared to do.

Thankfully, a perspective I can hope to make improvements from is the perspective that I hold: that of a software engineer and a science, technology & society scholar. I was aware that assembly processes were involved in the creation of software artifacts, but it was only as I worked on this project that I began to focus on them. These processes are involved, but not particularly meaningful, when software includes only the work of one author (or, more often, one author and the software project that produced the compiler / interpreter). However, as software artifacts grow to include artifacts (software or otherwise)from many different software projects, it seemed like there were real questions of agency involved.

If I make a software project and its products are used to do harm, how can I understand my role in that? Certainly, that a tool I created is turned to harmful ends is worth considering. The idea that developers' responsibility stops once they've released their work to the public is absurd. There are an enormous number of efforts, both at the software and legal levels, to ensure that software is used in ways that its creators accept (Gabriella Coleman, 2012; Kelty, 2008). There are few easy answers to be had, but as it stands there are few tools to help software developers assess how much their work has contributed to another.

The Goal of My Intervention

I decided to make a tool that would connect Git repositories and Python files using the Python import system. The tool traces a path from file to file, using import statements to connect them. It will try to record if an entire files is imported (as in 'import X') or if smaller portions are imported instead (as in 'from X import a, b'). After locating the relevant files, the program collects all of modifications associated with them recorded in the Git repository. Then it matches the identities attached to each modification to each file. Finally the tool assembles a list authors and a list of lines

[17] Part way through this project I, and most everyone else in the world, experienced the Covid-19 pandemic. As of this writing, the pandemic is ongoing. However, I have no way of appreciating how Eden would impact me were it mediating my access to supplies or services. I could not replicate that power relationship and it would be inappropriate to act as if I had.

of code in specific files that are contributing to the function of a particular file.

My hope in building this tool and not another was to highlight the material contributions of the work of various authors to a particular source file. This doesn't reflect anything close to all of the work that has contributed this particular file existing over all other possible files (there are many other factors: the cultural context of each particular files' creation, the immediate and long term goals of the functionality the file attempts to implement, etc). However, for every element of a software artifact, there is a concrete set of components that affect that elements' function and those components have a concrete set of authors. I hope this will be a useful starting point.

I named my the Python module containing my tool **materiality**.

Finding the Pieces

First, I should briefly explain the relationship Python modules and Python source files. The idea of a Python module is used to organize functionality within the Python programming language. A Python module contains Python objects (including, possibly, other Python modules). Each Python module contains at least one Python source code file. A Python module that does not contain other Python modules contains exactly one Python source file. A great deal of the work **materiality** does is connecting logical modules mentioned in Python source files to particular files in particular places in storage. As a result, I will frequently be talking about 'modules' or 'files' nearly interchangeably: the module being the logical representation inside the Python software artifact, the file being its material instructions stored on the computer.

I started by trying to use existing python libraries to scan Python code, but found that they weren't well suited to the task. The first library I looked in to, py2deps, was based on a command line tool for making graphical representations of the libraries used by a Python software artifact. This seemed very similar to my purpose, but it turned out that py2deps did not differentiate between using some or all of a python library. Its interest was detecting if a library was involved in any way and did not differentiate between using some and all of a library. I tried some other libraries, but none of them

seemed to match my desired feature set: finding import statements and connecting an import statement to one or more lines in the referenced file.

After being unable to find a Python library to do what I wanted, I decided to use the abstract syntax tree (AST) functionality in the Python standard library. A Python AST is generated by reading a file of Python source code. The resulting AST will contain all the Python statements in the file it read, but expressed as structured objects in Python. This capability was originally written to allow the processing and execution of Python source code by the Python software artifact, but it can also be used to help Python programs process Python source code for other purposes.

I used the AST module to parse each Python file I wanted to investigate and catalog three things within each file: the import statements, anything that ads a symbol to the symbol table (as they may be imported), and the line number of each statement in the file. Once I had code that could extract this information from any properly formatted Python file, I was ready to begin the real work of collating contributions from various authors.

The Algorithm

Briefly, here is the high-level algorithm **materiality** implements. The first Python file is provided to the tool and will be referred to with the name ⟨first_file⟩.

1. Create an empty index of python files named ⟨imported_files⟩.
2. Create a list ⟨files_to_scan⟩with ⟨first_file⟩in it.
3. While there are any file locations in ⟨files_to_scan⟩, do:
   (a) Remove a file location from ⟨files_to_scan⟩.
   (b) Build an AST (referred to as ⟨AST⟩) from that Python file
   (c) Add this file location to ⟨read_files⟩(a list of files I have already read).
      i. Create a new tracking *object* named ⟨file_info⟩
      ii. In ⟨file_info⟩, record:
         - Any *import statements* found in ⟨AST⟩
         - The *symbols*[18] generated by any ⟨AST⟩nodes, as well as what those symbols would contain.
         - The lines associated with each *symbol* found in the ⟨AST⟩.
      iii. For each *import* in ⟨file_info⟩try to find a particular file that it refers to:
         A. Try to locate a file using several techniques, not all of which apply to all *imports*:

[18] In computer programming languages, a symbol a generic term for a saved value. The name comes from the practice in many languages of using a symbol table, which is an index of variable name to current value.

- If the *import* is relative, navigate the file structure searching for the file it refers to.
- Ask the Python software artifact to locate the file for the *import*.
- Ignore *imports* that fit these descriptors:
  - *Imports* that come from the python standard library (I did not know how to account for elements of this code that exist in within the Python software artifact).
  - *Imports* which, when asking the Python software artifact to find them, returned an error.

B. If there is a copy of the Python file I located in a Git repository, use the file located within the Git repository.

iv. Add the file locations to ⟨files_to_scan⟩as long as they meet these requirements:
- They were not ignored as described above.
- A particular location could be found.
- This location is not already stored in ⟨read_files⟩.

v. If the file location was added to ⟨files_to_scan⟩do the following:
- Retrieve the list of changes and associated authors made to this file from the appropriate Git repository and add them to ⟨file_info⟩.
- Add ⟨file_info⟩to ⟨imported_files⟩.

4. Empty ⟨read_files⟩.
5. Create am empty tally of authors and change numbers named ⟨changes⟩.
6. Load the ⟨file_info⟩associated with ⟨first_file⟩from ⟨imported_files⟩.
7. Repeat the following for each ⟨file_info⟩selected:
   (a) Add this ⟨file_info⟩to the list of ⟨read_files⟩.
   (b) Add the stats associated with this file to ⟨changes⟩.
   (c) Examine each *import* in this ⟨file_info⟩:
   - If the *import* is not in ⟨read_files⟩yet, select it and repeat this process.
8. ⟨changes⟩now contains the list of authors and files which have an impact of the contents of ⟨first_file⟩.

This produces a list of as many authors as possible and generally characterizes their particular contributions to ⟨first_file⟩. As of right now, there are a number of flaws in the results and I would not recommend anyone use **materiality** without it being further developed[19].

Methodological Choices

There is not one obvious way to approach this and my tool reflects my choice to approach things one way over other ways. I want to briefly acknowledge and situate those choices.

---

[19] Also, **materiality** is currently sitting in the Eden repository for no particular reason other than I was using Eden as an object of analysis. The module name is "materiality."

First is that this tool collects changes over time and not simply the authors who are responsible for the last changes to each particular line in a file. This is an attempt to reflect the way that files change over time and are touched by many hands. Git will record the last person to change each line, but those changes are almost always based on a lineage of work that stretches back in the history of the file. On the other hand, this approach risks elevating substantial work from the distant past over recent and more impactful work. This concern is pervasive throughout the tool.

Second is that the tool ignores Python system libraries, though Python libraries are used throughout the Eden code. Python system libraries are libraries whose functionality comes both from Python source code and functionality within the machine code of the Python software artifact. I avoided counting them because I was unsure about how to account for that mixed responsibility. Including system code would mean fully grappling with cross-language material connections in a way I'm not prepared to do.

Flaws in Current Functionality

materiality is a flawed creation. I was learning about the details of how Python handles import statements, the Python AST and a number of other elements of the Python ecosystem. This project was also built in concert with the academic work to bring this software into conversation with STS literature over a relatively short period of time. The result is deeply flawed. I will briefly go over known problems.

The interaction of Python's system for referring to modules and my management of the Python symbol table has a number of problems. To explain I need to go into some detail about these two systems and how they interact.

A Brief Overview of Symbols. Symbol tables in Python generally follow patterns in other languages. Python maintains a list of all Python libraries, with each Python library taking on either the name of the file the library is defined in or, if there is more than one file, the directory the library is contained[20] in. Each Python file that

[20] In order for a directly to be a well formed Python module, it must contain a Python file named ___init___.py. This file may be empty, but may also execute Python code. ___init___ files are

the Python software artifact has seen is either recorded as a library, or (if that file is a component of a larger python library) as a sub-section of a Python library. This means that, for instance, my library is named *materiality* because that is the name of its directory. The `ast_crawler.py` file is stored under the *materiality.ast_crawler* label.

This system is used by Python to avoid redundant loading and processing of Python source code. The first time *materiality.ast_crawler* is encountered the Python software artifact loads and processes that file. On subsequent encounters Python refers back to its previous work.

A similar system is used to track objects *within* Python modules. Various Python statements (including import) add entries to a table of names and their associated Python objects. The behavior of this system can become complex, but at a high level it functions like the Python module system - symbol tables can be navigated using names separated by dots. For example, returning to *materiality.ast_crawler*, *ast_crawler* contains the Python object *ImportReference*, which in turn contains a value named *SYSTEM_LIBRARY*[21]. To bring this value into your python module, you could write `import materiality.ast_crawler.ImportReference.SYSTEM_LIBRARY`. This import statement causes the Python software artifact to navigate into a directory, select a particular file, then make two selections from two symbol tables.

Symbol Trouble. The current version of **materiality** frequently has trouble with symbol tables. If I understood exactly why, then I would have fixed the problems, but I can do my best to catalog them.

Some Python statements that add a symbol are simple to process: function definitions, object definitions. However, there are instances where symbol creating statements (such as assignment operations using the '=' sign) have complicated structures. For instance, one might write: `(a, (b, (c, d))) = function_call()`.

---

commonly used to make the elements of the library that are intended to be used by outsiders more easily accessible. For example, an ___init___.py file in the *example* module could contain the statement 'from .functions import run.' This would allow module users to type 'from example import run' instead of 'from example.functions import run.' This approach would allow *example* users to know less about the internal details of *example*.

[21] This constant is used to mark import statements which refer to Python system libraries.

There's an implicit structure to this statement that I can write out[22], but I was unable to write Python code that handled situations like this one. Doing so would require reference objects that stand in for the results of future Python software artifact operations. It's also worth noting this flaw would require moving beyond the assembly step, which I was not prepared to do.

Another problem is that, simply, there are instances where my programming errors have made it difficulty to navigate the shared file & symbol table structure I outlined above. There are many errors in the logs which indicate that my tool is unable to match an import statement against my version of the appropriate symbol table. I suspect many of these problems are caused by the fact that I don't (as Python expects), add things that are imported to the symbol table of the module they are imported into. For instance, the long import statement I used as an example previously (`import materiality.ast_crawler.ImportReference.SYSTEM_LIBRARY`) would add a `SYSTEM_LIBRARY` symbol to the symbol table of the module. The current version of my tool does not do this and many Python modules make use of this technique.

I believe that these errors minimally impact the results. **materiality** does not take into account duplicate references to the same file and I believe that most problems with how I handle symbols would not result in adding any new authors or lines to the overall totals.

However, the flaws in my work on symbol management and others which I will go over in a moment, indicate that my methodological approach when writing this program is flawed and the results should be treated as highly suspect. I believe they are still interesting and informative even in their less-definitive state.

Counting Errors. The initial "final" version of **materiality** was double (and more) counting files connected to the target of its analysis. I had written a system that was intended to, for a given symbol, return information only on *the section of the module containing that symbol*. In fact, much of my development time was spent trying build a system that would achieve that granularity. Unfortunately, it does not seem like

[22] In this case, if we refer to the value of function_call() as X, the symbol table would look like this: $a \Leftarrow X[0]$, $b \Leftarrow X[1][1]$, $c \Leftarrow X[1][2][1]$, $d \Leftarrow X[1][2][2]$

I succeeded (or I failed to properly collate the results). In any case, modules that were accessed through specific symbols inside the module (as opposed to importing the entire module / file) had the change counts for the entire module

Other flaws. There are other signs that I have made mistakes. Though the structure generated by this algorithm should be a true tree if it is constructed properly (import statements must form a tree), trying to navigate the entire structure results in infinite loops. I do not understand why.

Were I to do this again, I would build a clean internal representation that smooths the difference between the computers' file structure and a python objects' symbol table, but I did not understand I needed to do this early enough and the current code is a mess.

The procedures for associating a particular Python file with a particular Python likely could fail less often. However, without adding records of when and how they fail that are reported at the end of the process, estimating the impact of an improvement in this area is difficult. I also do not evaluate if import statements are used. Its entirely possible for a file to import a Python module that it does not use (I have unused imports in the work I did for this project). These can be detected, but I do not do it.

Finally, there were cases where overlapping errors collaborated to produce a functional software artifact. In one place I designed the system for processing *import* statements to mark an import as invalid if the Python software artifact could not locate a file for it. However, in another place, I ignored that marker and assigned it a file based on the module name the *import* referred to. Each system ignoring the other was an unintentional omission, but in combination they produced a functional system (albeit one that frequently falsely complains it can't find a module).

Material Connections of Validators

In some sense, the tool speaks for itself. It is, as is all software artifacts, materialized intent. There is a level of understanding of software projects and artifacts that can only be acquired through interacting with them. However, it's no fun to make

something and not show it off.

Validators by the Numbers. Below are the results of running **materiality** on the Python file[23]containing the validators that I repeatedly encountered while engaging the Eden. This file is responsible for checking that data inputs are in line with expected formats. The validator with the bug I tried to fix ensures an input it is given matches the format of a phone number. Some other validators defined in the file check that an input is the proper format for latitude, is a number or is a UTC date & time. Each validator builds off of and is connected to the the Validator object in the web2py project, which provides common operations used in accepting or rejecting user inputs.

Table 3
Result Summary

Authors Involved	119
Files Involved	86
Lines of Code	41,226
Total Change Count to Files	170,806
Longest File Active Lifetime	8 years
Average File Active Lifetime	Between 4 and 5 years

- The author count includes myself, because I (defiantly) merged my fix to the bug I found in the phone number validator (after changing it to work correctly).
- The count of files show here involved includes the `s3validators.py` file **materiality** was given.
- File active lifetime should be understood as the period of active work on that file. For example, if a file was created 6 years ago and last changed 4 years ago it would have an active lifetime of 2 years.
- These data are summaries of a larger dataset. An author-by-author breakdown can be seen in a section at the end of this paper and even more detail is accessible through a Google sheet whose link can be found in the same section.

These results highlight the collaborative nature of software development, especially FOSS software development. There are many authors and files that support the various Validator classes Eden has defined. In particular, it's interesting that these files contain

[23] The path inside Eden's repository is /modules/s3/s3validators.py

just over forty thousand lines of code, but over four times that number of lines have been added or removed from those files. This shows how code changes and evolves over time and highlights how software is not any particular version of the source code or software artifact, but a process and practice that occurs over time (Mackenzie, 2006).

Diving a little deeper into the data, it's possible to characterize the weight of the impact from the different projects on the set of validators defined in `s3validators.py`:

Table 4
Percent of Involved Changes By Project

Project	Location of Connected Changes	Connected Changes As % of Project
Eden	16.1%	0.44%
web2py	62.2%	3.30%
pydal	20.7%	9.23%
yatl	1.1%	77.75%

I think the most interesting thing here is how the work on the shared Validator class from web2py is reflected in this breakdown. The Eden team is able to do less work by relying on the work put into the object(s) they are extending (and the other objects inside web2py that support that foundational work). It would also clearly help if I had some method that accounts for project size - Eden is over four hundred and sixty times larger (2) than yatl, meaning that 0.44% of Eden is likely[24]more work than 77.75% of yatl.

What can't be Seen. I'll reflect more on this in my discussion on possible improvements, but I should note here the overwhelming flatness of my results. There is a myopic quality to looking at programming in this way. The act of displaying all the contributors together, without differentiation, is itself a D. Haraway (1988)-esq god trick. It took a good amount of work to get to this point - more than I'd like to admit and more than a good programmer would need. I believe there is worth in these results as a reflection of the exercise and the exploration, but they are a snapshot from a journey that may never be completed - not a triumphant demonstration of the ideas I've been speaking about.

[24] The reason I say likely is that I'm not sure how to compare the total number of changes in the entire repository (which include the non-Python code) to the language-specific line counts in 2.

Reflecting on my Travels

Software Exists Outside of the Social

Instead, I think what this shows is a material reality interacting with a social relationship. Each given version of Eden's source code has a particular material relationship with each version of each component it could use. These relationships have an existence outside of any social system because computer code can be executed mechanistically (Cox & McLean, 2013). The official / unofficial layer, where the team that writes the source code for Eden and sets forth the list of officially supported component relationships, exists on top of this material reality. But this social layer does not change what software *is*.

Understandings of software where only the official versions of Eden are "the software called Eden" lead to an understanding of software that defies common experience. It would mean that, once the Eden team announced that they were migrating away from python 2, all of the python 2 Eden software artifacts that exist and function now suddenly stopped being software. It would also mean that, because I have changed some elements within the Eden source code[25], my software artifact can no longer be Eden - even if I used all the official supporting components. It would require an impossible level of information about every component and every element of every software artifact that appears to be Eden.

Once we accept both official and unofficial software artifacts are that software, it becomes clear that another social understanding exists along side just the official and unofficial version of software. Most people who encounter a particular software project have no idea if they are interacting with an official or unofficial version[26] (or that such a

---

[25] Modifications to the Eden source code are also common. I've modified the eden source code somewhat and it is, I would argue, still Eden. Where one draws the line is questionable and, I think, worth studying in its own right.

[26] It is easy to example how someone might assemble an unofficial version of a piece of open source software like Eden. It may seem harder to imagine how the public gets access to an unofficial closed source piece of software that's distributed as machine code (like, say, the Facebook app). However, this situations are not as rare as you might expect. If you've ever owned a phone that is no longer receiving updates, or used a version of Windows that Microsoft no longer supports, every program in that environment is likely unofficial - because the companies no longer support that environment.

distinction even exists). So there exists this general social understanding about each piece of software (including Eden) that individual draw from their own experiences. These experiences might be with official software, unofficial software, or even be based on a mis-identification of the software (such as receiving a counterfeit product and leaving a bad review for the original) (Suthivarakom, 2020).

It's useful to draw out these different social experiences of software because it allows us to think in a more informed way about the impact of materialized perspectives. It is not just the source code of a particular software project that adds to its behavior, it is also the various components that source code is combined with. For those who only interact with post-assembly software artifacts, the relationship of the software they are using to the official software is unknowable. Software that has the reputation for being unstable and unreliable can get that reputation because people are, knowingly or unknowingly, using pirated copies that crash because of anti-piracy code (Fitch, 2008). The team that makes that piece of software may never know why people think its unstable, because they would need to understand the assembly processes that particular software artifacts went through. The users may want the software to be different, but though the software team is the single entity most responsible for how a particular software artifact functions, they may simultaneously be unable to understand or repair the problem.

Implications of my failed bug fix

Before going into the interesting aspects of the König quote, I want to talk about something I find interesting about the context of how the discussion about the bug took place. This was, as I said, a discussion between people who both had access to one (or more) Eden software artifacts. I originally noticed the bug when I was interacting with my software artifact. This is in line with how most software projects come to have an impact on the world: through directly or indirectly affecting the world through the software artifacts that the project produces or makes possible (Kitchin & Dodge, 2011). While software impacts our lives in direct and indirect ways, those impacts derive from

how one or many software artifact(s) interact with particular human agents. Though software has made is possible to remove humans at many of the intersections of power, it is still expressed through a multitude of relational contacts between a representative of power (a person or a software artifact) and a human outside that power Cheney-Lippold (2018); Deluze (1995).

Do Software Artifacts Matter. Given all of that, I think it's interesting to note that the entire discussion about bugs took place without referencing the behavior of any particular software artifact. I've already speculated that the Eden project may recommend older components to minimize the risks of improper assembly and that questions *about* the correctness of assembly are common (trendspotter et al., 2020). Because investigating the particulars of a misbehaving software artifact are common, the lack of discussion here is interesting.

When a software artifact is under discussion, that discussion often proceeds with the implicit goal of finding a mistake or a mismatched relationship in the assembly process. My bug fix, in contrast, was to a particular pre-assembly component. In order to realize my change, a new software artifact would need to be assembled. Nothing in my attempt to fix the bug or in the eventual official bug fix can repair existing software artifacts (König, 2020; König & Drexler, 2020). This explains the lack of interest in any existing software artifact - nothing about any software artifact that might or might not express this bug is of interest. Instead, when the bug is fixed in a component of Eden, any potentially problematic software artifact will be replaced with a new one assembled from updated components.

To underline this point again - software artifacts are never updated. When a 'patch' is applied to a software artifact, the goal of that patch is to make the existing software artifact identical to the software artifact that would be generated by a fresh assembly. This means that a patch program is a new software artifact that will effectuate a different assembly process than usual with the goal of producing the same outcome as a full re-assembly with all components (Endsley, n.d.). It's also common for a software artifact to contain many internal software artifacts and for a patch to replace

one or more component software artifacts in full. However, the software artifacts that accomplish these tasks of updating existing software artifacts are separate software artifacts in their own right. A software project can choose to utilize them or not utilize them, but their choice won't change how they "fix" problems found in an existing software artifact.

The bug was fixed by changing a component with the intent of that component being assembled into a new software artifact. It could also have been repaired by creating a new software artifact to intentionally make changes to existing Eden software artifacts, but this would need to be repeated for every previous software artifact[27]. So software artifacts are always replaced, because digital technology makes it easy to do so and the practical consequences are the same either way: a particular number of bytes in a particular order on digital storage.

This means that scholars must be careful when applying concepts whose identities were formed based on our collective experiences with physical goods. It's true that software is maintained, updated, repaired, cleaned, polished, and so on. Those operations often serve the same purpose as similar operations on physical tools, but the material character of how they are carried out will be dramatically different. This will impact the worlds of the people doing the work, the particulars of how these operations change the operation of software, and the proper weighting of concerns in assessing the impact of these operations. Because the material that composes the physical character of software artifacts is cheap and plentiful (electrical charge), many of the previous systems that evolved in response to more complicated systems of material construction have limited implications for software projects and software artifacts.

---

[27] There is an important caveat to this claim for interpreted languages: their elements often can be replaced piecemeal. It would be simple to, for example, replace the file which was altered when I fixed my bug with another file. The rest of the software artifact would remain untouched. Still, interpreted code is rarely updated piecemeal. The HTTP protocol, through which most javascript applications are distributed, does not have a way of updating cached (previously distributed) programs (MDN Contributors, 2020). It may be that using replacement instead of change distribution may have once been driven by technical concerns, but in the age of relatively inexpensive data and storage those concerns are gone. The reason to continue using replacement is cultural instead of technical. This is why, even though interpreted languages could distribute changes more easily than compiled languages, they use replacement.

The Limits of Developer Attention

I want to briefly expand on the idea, expressed by König, that testing in too much detail can harm software development. I think this idea, in combination with some discussion of assembly in the following section, will be helpful for thinking about agency in software. An uncharitable understanding of the idea that too much testing harms development is that developers might not care about some dysfunction, but I see no evidence of this. The Eden project immediately implemented a fix for the problem I found, even though they do not wish to add a test for it.

Instead, I think this approach reflects a Citton (2017)-esq approach to the problem posed by the enormous complexity of software artifacts. In just the four libraries I focused on for this project there are over five hundred and fifty thousand lines of code2. The version of Eden I have been working on has seventy two authors (exuding myself). Even if each developer was contributing equally (which they are not), this would require each developer to test every single outcome of around seven thousand lines of code. This would be more code than many contributors have added to the project themselves. Without drawing conclusions about the degree to which developers have a responsibility to monitor behavior of their software artifacts, it seems very likely that it would be impossible for them to monitor all possible behaviors.

The implication of this is not to let developers off the hook because monitoring all behavior is impossible. Instead I want to bring software development back to the social context that drives it. It seems like the Eden project is focused on how their components interact with data rather than how they interact with people. That is to say, they would rather create a software artifact that gives the user incorrect information about its actions, but records the information that the developer expects. This runs the risk of becoming disconnected from user experience of Eden[28].

This reflects a focus on software artifact behavior that is inaccessible to users. It's true, to a certain degree, that users can be confused by particular elements of how a software artifact is functioning and still have an overall impression of what 'it is doing.'

[28] I have no reason to believe this has occurred.

However, the general impression of what Eden (or any software project) 'does' isn't transmitted directly from the understanding of the software project. Instead, those impressions are formed from users' many experiences with particular software artifact in particular situations (Boellstorff, 2015; Eubanks, 2018; Schüll, 2012). Often, users do not even have to interact with a particular software artifact to form an impression of its impact (Kitchin & Dodge, 2011). The experiential nature of how impressions of software are formed means that users are unlikely to change their impressions of the software because one or more message is wrong, but it seems certain that at some point users will feel misinformed to such a degree that their impression changes.

The other risk is that the material qualities of the software and the messages it sends to its users begin to diverse in unsustainable ways. Developers can choose to have their software artifacts send messages to users that don't reflect the internal state of the software, but this will likely make continuing to operate the hidden aspects of that software artifact (and project) difficult (Bivens, 2017). Producing a software artifact that has external representations that aren't supported by all of its internal state means severing internal relationships and creating parallel internal functionality.

How Would the Left Hand Know what the Right Hand Does? All forms of testing are forms of surveillance. This makes testing practices vulnerable to the shortcomings of surveillance[29]. It also reflects that software projects recognize that their software artifacts may not behave as they would wish. Surveillance systems are systems of control that are applied in situations where control is not certain. This suggests again that the expression of the agency of a software project through their software artifacts is not straightforward. The practice of surveilling software artifacts through testing speaks to the potential for intent to be subverted through assembly.

---

[29] An expanded survey of literature on surveillance would be extremely useful here.

What does Working Mean Here?

I want to return to talking about the various experiences of functionality and non-functionality I've encountered in this work [30]. First I'm going to try and explore the ideas of assembly and then how to think about the functionality of assembled artifacts.

Assembly is a process whose limits are challenging to clearly mark. There are instances where assembly is clearly successful, such as when a software artifact works correctly in all ways or in instances where the failure of an assembled software artifact clearly comes directly from a single flaw in a single component (such as the international number bug). There are also limits where assembly is clearly unsuccessful: when a component cannot be found, when a language that must be translated into machine code (or any language) is found to contain a syntax error (Cramer, 2008). However, when I created a software artifact that failed to work because of a broken relationship between its components, I began to wonder if my definition of assembly was flawed.

Unlike compiled languages, which fully parse their constituent source files before reaching a state where an executable software artifact exists, Python software artifact performs assembly immediately before executing a python program[31]. In fact, the python assembly is piecemeal - each file is only read when needed, so files that are part of the software artifact could go unused for any period of time before being used in the software artifact. It would be possible to, for instance, begin running an Eden software artifact and replace the line of web2py code that, if it were executed, would cause a crash. If the code was replaced before the Eden software artifact called the code with the bug, no crash would occur. So do I need one approach for assembly with interpreted

---

[30] To be blunt, I find the common ideas of 'working' or 'broken' to not be that useful and suspect that a deeper literature review would have helped me to speak clearly on this question.

[31] It's worth expanding a little how this works. The python software artifact executes the python in the initial python file provided. This could involve executing statements immediately or it could involve defining programmatic objects that may be invoked later. The 'import' statement is how, in python, you note you want to include the result of executing another python file. When a python software artifact encounters an import statement, it executes the associated file. This means that many python programs have an initial import chain of stand-alone import statements. However, import statements can also be located in the programmatic objects that are defined for later use - and those objects will not be executed unless they are invoked elsewhere. So assembly is a somewhat fractured process with Python and other interpreted languages.

languages and another approach with compiled languages?

No, because the apparent differences are illusions. Compiled programs can also have their contents changed mid-executions (this is the usual method for 'breaking into' a computer). Changing the contents of a program that has already been translated to machine code is simply much less convenient than changing the text in the source file of an interpreted language. The moment of failure that I encountered when web2py tried to invoke Eden code it no longer knew how to find was a moment where the computer tried to follow a chain of relationships. It would not be possible to know if any given instance of an Eden validator that inherits from web2py code would work correctly without running the program. This means that finding the error would require solving the halting problem, which is impossible (Kaplan, n.d.).

For the purposes of this this project, what I call assembly is every aspect of preparing a program to execute that can be done without needing to actually execute the program (i.e. the limits of the halting problem). These processes (compilation, the locating of python files) are often themselves executed by software artifacts. As long as the work being done could be done without executing source code, it is considered part of the assembly step[32].

Why is this important? Does it matter for sociologists when assembly stops and execution begins? The end of assembly is the point where human intervention into a software artifact largely stops being useful. After assembly is complete, the balance of which components will impact the overall behavior of the software artifact is fixed. The only human intervention that is possible is the decision to start or stop the software artifact[33]. This is doubly important because our ability to predict what a software

---

[32] When Python import statements are causing new files to be read and executed by the interpreter, we are in a kind of mixed execution / assembly phase. The Python is doing a mix of adding symbols to the symbol table (which other languages do during compilation) and executing program statements. If the Python software artifact were to fail to find an import or encounter a syntax error, that is an assembly error. If executing a program statement causes an error, it was not in assembly. This means that Python may translate some python statements to machine code, execute them, then fail when it reaches another assembly stage. I do not believe this is problematic for my work.

[33] A software artifact can be designed to seek input at any point, but this does not change the fact that humans cannot *intercede and change* the actions the software artifact will take. The list of actions the software artifact takes may involve seeking human input, but the possible range of actions is already fixed. As Mackenzie (2006, p. 181-182) points out, once agency is fixed in software, the software will

artifact will do when it is executed is hampered by both practical and theoretical limits (Kaplan, n.d.).

This reality is reflected the practice of testing in the development of software artifacts. The Eden software artifact assembled in Eden's test code is not exactly like the final software artifact (it largely differs in the database entries that compose it), but the goal is to make it as close as possible to the final software artifact. For instance, the assembly error I encountered with web2py was easily detected by tests. Testing is a reaction to the fixing of agency in a software artifact and an attempt to ensure that the agency reflected in the final software artifact is sufficiently similar to the developers' intent. These tests can take the form of source code that is incorporated into one or more testings software artifacts or in planned interaction with the final software artifact before releasing it to the public.

Reflecting on my Intervention

Foreseeable Improvements

I will also briefly catalog things I think could be improved about this work. This will both sooth my frustrations and mark what directions the **materiality** software project, as it exists today, could be taken in.

Deeper Engagement in Git. The current version of **materiality** engages with modification history recorded in git in a very shallow manner. It crudely counts changes to files without reference to content. It does not detect when a file has been moved. It can't tell the difference between a truly new file and a new file that is created by removing some of the content of an existing file. Any time source code is copied or moved its history is lost. Not all of these scenarios are clearly detectable - text matching can be difficult and uncertain - but an effort could be made.

Change Distance. The most obvious improvement by far would be for **materiality** to characterize the distance of the various impacts in some way. At the

---

reflect it in its execution. Any opportunity to "give feedback" to an assembled software artifact will, at best, select another path from the possible paths fixed by assembly.

moment, if any file can be reached through the tree of imports leading out from the one **materiality** starts from, its authors and change counts are added to the list without adjusting for the distance. This means tallying work directly on the file of interest and work on, say, a file used by a related library to print error messages would be displayed in an undifferentiated way. The current software project tosses all change counts and authors into a single category, which is obviously wrong.

Bringing a sense of distance, or varying amounts of weight, would also be an opportunity to apply the sociological theories that form the theoretical background for my thinking in this project. Both linking together various Python files and attempting to weigh the relative impact of those links recalls Mackenzie (2006)'s work on how relationality structures software projects. It also reminds me of D. J. Haraway (2016)'s notion of "tentacularity" - the relentlessly interconnected nature of the world and how that wealth of connections to enliven our sense of the possible. It's also clear that the way **materiality** works now - making no effort to characterize how any particular change is related to the file being analyzed, is the kind of 'god trick' that Harding (1992) and D. Haraway (1988) set out to critique. I have created another view from nowhere.

I do not know now what the best way to characterize the disparate contributions from changes at different points in the tree of related files. Usage analysis could be a large improvement. Import statements add objects to to the module symbol table and **materiality** could analyze the usage of those objects. Carrying this practice across the entire import tree would highlight links of action (as opposed to an undifferentiated link). As noted previously, however, such an analysis might be deceptive. There are real limits to how much static analysis of computer programs can predict the behavior of those programs Kaplan (n.d.). As much as the current program is a view from nowhere, I want to be mindful of the possibility of rendering a false view.

Ways of Seeing. The current output of the tool is, to put it bluntly, opaque. Quoted below is a sample output from the *adapters* module in the *pydal* module. This is not the final output, but is as close as the software artifact can come to displaying the web of connections.

```
</Users/ddrexler/src/python/web2py/gluon/packages/dal/pydal/adapters/ __init__.py>
SM[pydal.adapters]-> Module[]
  Imports:
    [1]Import <!S>re
    [2]From <+R>pydal._gae Import gae
    [3]From <+R>pydal.helpers._internals Import Dispatcher
    [71]From <+R>pydal.adapters.base Import SQLAdapter,NoSQLAdapter
    [72]From <+R>pydal.adapters.sqlite Import SQLite
    [73]From <+R>pydal.adapters.postgres Import Postgre,PostgrePsyco,PostgrePG8000
    [74]From <+R>pydal.adapters.mysql Import MySQL
    [75]From <+R>pydal.adapters.mssql Import MSSQL
    [76]From <+R>pydal.adapters.mongo Import Mongo
    [77]From <+R>pydal.adapters.db2 Import DB2
    [78]From <+R>pydal.adapters.firebird Import FireBird
    [79]From <+R>pydal.adapters.informix Import Informix
    [80]From <+R>pydal.adapters.ingres Import Ingres
    [81]From <+R>pydal.adapters.oracle Import Oracle
    [82]From <+R>pydal.adapters.sap Import SAPDB
    [83]From <+R>pydal.adapters.teradata Import Teradata
    [84]From <+R>pydal.adapters.couchdb Import CouchDB
```

This form was designed to help me understand what the program was and was not finding. The numbers in `[]` are line numbers in the `__init__.py` file. The symbol and letter inside the `<>` symbols indicates the status of the import. `<!S>` signals that this import refers to a system library and will be ignores. Import statements with a `<+R>` mark are relative imports whose referent has been successfully found[34].

However, many different graphical displays would be possible. The file itself could be laid out and sections of text connected to the tree of python files that influence that texts' behavior. **materiality** could generate graphical representations of the networks of Python modules involved. Instead of focusing on the files themselves, the collaborations between authors could be highlighted. Each file could seen as moving between contributors over time and **materiality** could render a graph of how the feature being analyzed had been built by humans in specific ways and at specific times in the past.

---

[34] For example, the import on line 2 was originally written "from .._gae import gae", the program has checked that this file can be found and changed from a relative path to an absolute path within the *pydal* module.

Returning at last to Software

I opened this paper and this project by asking how assembly mediates agency in software. The literature is clear, and I agree, that there is a strong connection between software project intent and software artifact behavior. The Eden project describes their mission as "sav[ing] lives by providing information management solutions that enable organizations and communities to better prepare for and respond to disasters" and nothing I found in the Eden source code or the Eden software artifact suggests that the impact of Eden follows their ambition (Sahana Foundation, n.d.-a). The complexities of software development do not change the previously established connection between tool impact and creator intent Subramaniam (2014).

I also found, at every turn, material qualities of software that threaten to escape the ability of creators to detect and tools to prevent. One consequence of this is that software developers are expected to trigger and manage the assembly processes for any software artifact they wish to work on. The experience developers derive from that is key to learning about the components used by the software project in the software artifact, but it redoubles the limited view already embodied through software project surveillance of software artifact (**The Limits of Developer Attention**). Any developer who engages with the components (source code, software artifact libraries) and the software artifacts of a software project is constantly experiencing suggestions about the nature of the world. Within König's idea that "testing too much detail can add excessive rigor," I see a world view that the correct way to imagine and survival software is attending to a particular set of behavior over all others (König & Drexler, 2020).

The Eden project and the Eden software artifact is not equipped to detect deviations in behavior they are not watching. Once a software artifact is sent out into the world, its behavior is fixed. Problems discovered within it will need to be worked around or the problematic aspect ignored until it is replaced. Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s concept of blackboxes rested on the work of making complexity socially invisible, but that's not exactly what I see here. In

both **What does Working Mean Here?** and **Software Exists Outside of the Social** I encountered an object whose behavior defied shared social understandings. It was not just that my Eden software artifact 'broke,' but that it broke in a way that was particular and limited way. In the case of the phone number message, the failure fell outside the sight of the software project. The other failure was cause by my choices in assembly. Both failures could only be understood by looking inside the assembly process but, critically, not inside the particular software artifacts. Unlike a blackbox, whose contents become socially visible[35]when they fail, software artifacts keep their secrets. I think I need an updated idea.

Composites

Software artifacts are not blackboxes, they are **composites**. Composites are fixed assemblies of more than one component which inherits the material qualities of its components, mediated through the structure which those components are assembled into. Once assembled, a composite does not require human intervention. Composites are typified by extremely high internal complexity and complex behavior. Once assembled their possible actions are set. Those actions cannot be predicted by examining a composite. Composite cannot be changed, only replaced. Each component of a composite may be an instruction or a datum, but they are generally composites themselves. Generally, each component makes a proportional contribution to the material qualities of the composite it is integrated into, but layers of assembly can place components in unexpected relationships with unforeseen outcomes.

The identity of a composite derives from its behavior, not its label. The different software artifacts of Eden are not similar because they share the label "eden," they are similar because they are largely constructed from the same components. As Eden evolves and changes over time, its social identity will "stretch" to reflect a diversity of

[35] To elaborate on this theme, Latourian blackboxes do not actually prevent people from seeing what is inside them. He is instead trying to speak to why people *can* operate *as if* the internals of a blackbox are invisible. The fact that blackboxes contain other blackboxes goes to show that everything within a blackbox has a social identity that people can and will recognize and interact with if they are shown it. software artifacts on the other hand, have their internals transformed from an object that some can understand (source code) to something nearly totally opaque to everyone (machine code).

behavior, but that same stretching betrays that multiple objects with different qualities are sharing a single label. Eventually composites that share many, but not all components will be given separate labels or face a collapse in the usefulness of the label Bivens (2017).

Composites are immensely useful objects, as they do not require human intervention or labor to carry out their intended purpose. If the required material resource are made available to the composite, it will continue until it exhausts its resources or is stopped. Humans may set them up without any knowledge of their internals and without easy insight into the details of their function. This makes them powerful conduits for systems of control and surveillance.

However, due to their complexity, composites' behavior is unpredictable. How changes made to a component will appear in the behavior the assembled composite is difficult, sometimes impossible[36], to predict. Multiple components or layers of assembly can overlap in confusing ways[37]. Composite makers respond to this quality by implementing systems of surveillance to observe and verify their own creations. These systems of surveillance often involve composites with many overlapping components whose purpose is to observe internal behavior and report on internal behavior. Assembled composites may have similar self-surveillance functions, though like all functions, these qualities run the risk of accidental subversion.

Composites in Society

Scholars have noted specific situations where composites have, intentionally or unintentionally, acted in problematic and damaging ways (Cheney-Lippold, 2018; Eubanks, 2018; Schüll, 2012). Previously, the focus has been on the various projects (software and otherwise) that are creating and placing composites in positions of authority. This is important work, but it elides the social qualities that composites do

---

[36] For software, situations where prediction is impossible are quite common (Kaplan, n.d.). There is nothing that requires composites to be software, per-se, however, and any sufficiently complex mechanistic assembly would meet this requirement.

[37] See **Other flaws**.

not share with previous systems of control: their opacity. The qualities of a composite can only be discovered by interacting with and observing it.

Situated knowledges was motivated by the harm caused when we severe an idea from its creator. It highlighted that whether or not ideas are true *in essence*, they first must *seem* true to the people who 'discover' them (D. Haraway, 1988; Harding, 1992). The same is true of composites: the behavior that creators allow must pass through those creators' personal experiences. This process centers around the testing of some finite number of material qualities. Even though we can (and should) speak the general cultural values of software project, all of the experiences that would lead someone to believe a composite composite fits within a particular cultural world comes back to that composite producing a particular set of outputs for a particular set of inputs. How a composite behaves is set once it has been assembled. Placing a composite so it is properly situated amongst the influences that drive it can make its influences socially visible, but the composite will still *behave* the same way.

So any social engagement with composites must reckon with two truths: our experience of them is mediated by our social experiences and composites are machines. A composite may be re-contextualized, re-interpreted, re-situated or (most commonly) replaced. It cannot be changed. Each composite is created in a particular social context and that social context is generally reflected in how that composite is deployed by its creators (Mackenzie, 2006). But that composite is not trapped within that social context. If the outputs the composite produces for a given input seems useful in another cultural context, it may be adopted as a component. But this adoption does not take one part of the composite. It takes the entire thing. Including any behavior its original creators missed, or behavior its new adopters have misinterpreted. Composites are mechanical things masquerading as objects of culture.

This means we have a tricky path to navigate with composites. As I noted in **Investigating Assembly**, small changes can have large impacts on composite behavior. The behavior of a composite cannot be reduced to the behavior of a particular component. Relationships between components do not to be enabling, they

can be inhibitory. So changing any element of a component of a composite can, upon re-assembly, be shown to have set off a chain of other changes in other components and, ultimately, composite behavior. Tracking down the exact source of misbehavior is possible, but time consuming. The Eden composite is entirely open source - I could investigate any element of any component. That luxury is not typical. Many of the composites that are used as components are closed source and are protected from modification by copyright law. This means that flawed component composites are rarely, if ever, fixed directly. Instead, a new element is added to the assembly process to inhibit the misbehaving component or take over from it in certain situations. Every composite is a patchwork quilt of immediate solutions to problems of enormous complexity (a quality they share with many socially important objects).

Living With Composites

All of us are already living with composites every day. Even on days when we don't see an obvious composite, they are now deeply involved in the management of our world. They impact the space we move through even if they are not present (Kitchin & Dodge, 2011). One might imagine that the deceptive and unpredictable nature of composites would make them overwhelming and confusing, and reports of being overwhelmed and confused by software are common enough. At the same time, the idea of software as objective (like the idea of facts as objective) persists.

I suspect that is because we were prepared for the complexity, opacity and unmarked agendas that composites display by another common feature of our world: other humans. In many ways, I think composites are the closest we have come to realizing the dream of creating a new kind of creature. As many speculative fiction authors have pointed out, the results are often nightmarish, but they are also deeply compelling. Composites have enabled people to accomplish many tasks that once seemed impossible. The relative value of Wikipedia or Google or Facebook is open to debate, but there is an awesome material force to the way humans have used composites to transform the way many live their lives.

I fear humanity is sleepwalking into a complicated future. Complex times demand complicated tools to meet them and I do not think we should (or could) abandon composites. That does not mean we can afford to ignore their specific qualities and how those qualities may lead to composites suddenly, unexpectedly diverging from previous behavior. We must speak honestly about accepting limits on our own ability to understand the things we've created and ensuring that composites are placed in situations where their agency is limited and mediated by human actors.

Glossary

assembly process All software must be assembled. In the simplest case this means the placing of machine code instructions, one after the other, in computer memory. In more complicated processes, this can include the translation from one or more programming language to the appropriate machine code, linking libraries together, retrieving source code from a remote location, and so on. This is rarely a manual processes, and is often done through one or more software artifacts. How a software artifact is assembled has an impact on the behavior of the artifact. Components can be placed in such a way that they do not function as expected, that their contribution to the artifact subverts expectations (an artifact might include a component designed for machine learning but only use its error formatting functionality) or in a way that makes functioning impossible. Assembly processes are often managed by a software project with the aim of producing their software artifact(s), but the process should not be considered a component of the artifact. It impacts the artifacts' structure, but isn't carried around in the artifact.. 15

composite Composites are fixed assemblies of more than one component which inherits the material qualities of its components, mediated through the structure which those components are assembled into. Once assembled, a composite can operate without human intervention. Composites are typified by extremely high complexity and extremely complex behavior, but once assembled their possible actions are set. They cannot be changed, only replaced. Each component of a composite may be an instruction or a datum, but they are generally composites themselves. Generally, each component makes a proportional contribution to the material qualities of the composite it is integrated into, but layers of assembly can place components in unexpected relationships with unforeseen outcomes. . 5, 6, 55, 56, 57, 58, 59

python Python is an open source, interpreted, high-level, general-purpose programming language. It was initially released in the early 1990s, but has been continuously updated since then. Python 2 was the standard from 2000 to 2008, but in 2008 Python 3 was released with backwards-incompatible changes. This split the Python community, leading to Python 2 and Python 3 having slightly different versions of tools available. Python 2 continued to receive support and updates until 2020, when official support for Python ended (Wikipedia

web2py An open source web framework for Python 2.7 and 3+.. 23, 24, 27

Web Framework A library (or set of libraries) that handles translation from and to the language of HTTP and other web technologies. The library of functionality they provide can range from wrapping basic text in a format a web browser will understand to providing functionality that allows users to log in, receive real-time messages and interact with the same web site from multiple clients simultaneously (Statz, 2010). Each Web framework also has its own philosophy about what information the developer is expected to provide and how much the developer needs to set. Some frameworks attempt to provide everything that is required to build almost any website, while others focus on providing a core of functionality and may not include support for connecting to a database or managing logins. Generally, any web framework may be used to make any website, but framework selection will dramatically change the experience of the software engineers developing that web site.. 19

web framework A heterogeneous set of software that allow a programmer to efficiently write and manage providing a service over the Internet. This could be a website or a mobile app (often it is both) or a go-between for other services. Examples include web2py and Django in Python or Phoenix in Elixir.. 18

Acronyms

Eden Emergency Development ENvironment. 6, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 36, 37, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 54, 55, 58

FOSS Free Open Source Software. 6, 8, 9, 16, 17, 18, 22, 27, 41

ICT Information and Communications Technology. 15

Tables

These tables are missing some information that is available in this google sheet: `https://docs.google.com/spreadsheets/d/1fi5cNCA1mQgS8gZ7VL7MfalSXNQoD2pM1vC3uH18cNw/edit`. Additionally, the contributor who is identified as 'sherdim' used their real name, but I was unable to get LaTeXto display their name properly, so I have used their github handle. Their profile can be found here: `https://github.com/sherdim`.

Table 5
Eden Contributors

Name	Related Changes	Total Changes
Ashwyn	23	7,580
Aviral Dasgupta	88	6,147
biplovbhandari	6	26,692
Daniel Drexler	69	10,342
Dominic König	14,070	1,940,890
Fran Boon	12,314	3,816,361
Graeme Foster	236	241,090
hitesh96db	4	13,588
James O'Neill	4	4,038
Kunal Hari	11	8,920
Michael Howden	313	143,771
Pratyush Nigam	104	290
raj454raj	1	10,917
redsin	278	25,265
tirgil	2	3,804
VishrutMehta	2	1,153

Table 6
pydal Contributors 1

Name	Related Changes	Total Changes
abastardi	9	15
alan	2	11
boa-py	2	2
BuhtigithuB	670	5,524
Cássio Botaro	2	2
Christophe Varoqui	2	2
Dan Feeney	11	11
David Orme	30	30
Dominic König	13	43
dz0	1	1
Emmanuel Goh	4	4
Fran Boon	3	21
Francisco Tomé Costa	4	4
gi0baro	14,046	69,785
Giovanni Barillari	6,112	8,240
ilvalle	1,968	37,158
Jack Kuan	7	7
JusticeN	15	15
jvanbraekel	40	40
kvanzuijlen	2	2
Leonel Câmara	191	896

Table 7
pydal Contributors 2

Name	Related Changes	Total Changes
Martin Doucha	1,992	3,037
maxcrystal	11	11
Massimo DiPierro	6,320	903,691
Michael Loster	2	2
Michele Comitini	52	5,975
nikakis	18	18
niphlod	816	30,396
preactive	2	2
Remco Boerma	5	5
Richard Boß	5	5
rodwatkins	14	14
Stephen Rauch	2,852	4,993
Stephen Tanner	7	43
Tim Nyborg	6	6
Tom Stratton	9	9
Victor Salgado	12	18
Vinyl Darkscratch	2	49
Wanderson Reis	5	5
willimoa	7	64
xuyangbocn	2	2

Table 8
yatl Contributors

Name	Related Changes	Total Changes
Carlos Cesar Caballero Díaz	17	17
Massimo DiPierro	1,689	2,089
PhanterJR	127	263

Table 9
web2py Contributors 1

Name	Related Changes	Total Changes
abastardi	27	153
Adam Bryzak	2	2
Alexander Zayats	8	8
alexdba	17	17
Alfonso de la Guarda Reyes	19	63
Anssi Hannula	10	12
Batchu Venkat Vishal	2	2
Carlos Cesar Caballero Díaz	34	3,136
Carlos Costa	13	236
Cássio Botaro	22	207
cccaballero	10	138
Chen Rotem Levy	8	309
Chris DeGroot	4	4
Chris Garcia	12	34
clach04	45	45
Daniel Libonati	2	9
Denis Rykov	4	4
Dinis	4	470
Dominic König	2	10
Donald McClymont	7	7
Erik Montes	2	147
Fran Boon	11	28
geomapdev	153	223
gi0baro	12,022	38,807
Giovanni Barillari	65	346
hectord	6	6
ilvalle	1,019	65,421
Jack Kuan	8	24
Jan Beilicke	13	13
Jan M. Knaup	12	12
Jaripekkaf	56	56
Jeremie Dokime	14	54
Joel Rathgaber	22	22
Jonathan Bohren	57	59
Jonathan Vasek	2	2
Jose C	1	1
jvanbraekel	62	108
kelson	35	306
Kiran Subbaraman	88	99
Koen van Zuijlen	4	8
Kristján Valur Jónsson	2	2

Table 10
web2py Contributors 2

Name	Related Changes	Total Changes
Kurt Grutzmacher	352	2,378
Leonel Câmara	1,236	14,051
Lisandro	8	14
Luca de Alfaro	2	2
Martin Doucha	51	99
Massimo DiPierro	76,420	825,014
Mathieu Clabaut	24	88
mcabo	28	28
Michele Comitini	1,284	6034
mictee	33	71
Mirko Galimberti	11	52
mpranjic	6	36
Nik Klever	7	45
niphlod	6,304	45,996
Oleg	41	43
Omar Trinidad Gutiérrez Méndez	7	321
omniavx	2	2
Oscar Fonts	16	585
Oscar Rodriguez	66	66
Paolo Caruccio	6	475
Prasad Muley	23	23
Radu Ioan Fericean	30	66
Ricardo Pedroso	108	263
Richard Vézina	5,135	17,379
samuel bonilla	4	93
Scimonster	4	6
Seth Kinast	10	18
spametki	473	3,404
Stefan Pochmann	114	114
tiago.bar	2	4
tim	2	2
Tim Nyborg	43	109
Tim Richardson	222	337
viniciusban	18	261
Vinyl Darkscratch	150	121,159
winniehell	5	5
zvolsky	11	1,543
sherdim	2	14

References

Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge, MA, USA: MIT Press.

Agile Alliance. (2015, December). *What is test driven development (TDD)?* `https://www.agilealliance.org/glossary/tdd/`. (Accessed: 2020-6-5)

Bivens, R. (2017, June). The gender binary will not be deprogrammed: Ten years of coding gender on facebook. *New Media & Society*, *19*(6), 880–898.

Boellstorff, T. (2015). *Coming of age in second life: An anthropologist explores the virtually human*. Princeton University Press.

Canonical. (2017, March). *LTS.* `https://wiki.kubuntu.org/LTS`. (Accessed: 2020-5-7)

Canonical. (2020, April). *Releases.* `https://wiki.ubuntu.com/Releases`. (Accessed: 2020-5-7)

Chacon, S., & Straub, B. (2014). *Pro git*. Apress.

Cheney-Lippold, J. (2018). *We are data: Algorithms and the making of our digital selves*. NYU Press.

Citton, Y. (2017). *The ecology of attention*. John Wiley & Sons.

Cox, G., & McLean, C. A. (2013). *Speaking code: Coding as aesthetic and political expression*. MIT Press.

Cramer, F. (2008). LANGUAGE. In M. Fuller (Ed.), *Software studies: A lexicon* (pp. 168–174). MIT Press.

Dean, J. (2010). *Blog theory: Feedback and capture in the circuits of drive*. Polity.

Deluze, G. (1995). *Postscript on control societies. I deluze, negotiations, oversatt av m. joughin.* New York: Columbia University Press.

Di Pierro, M. (2020). *The database abstraction layer.* `http://www.web2py.com/books/default/chapter/29/06/the-database-abstraction-layer`. (Accessed: 2020-6-6)

Drexler, D. (2019, March). *Hack the planet.* `https://medium.com/@aeturnum/hack-the-planet-aaa4abc23bc8`. Medium. (Accessed: 2020-4-18)

Drexler, D. d. (2019, March). *What we lose when we lose gender.* `https://medium.com/@aeturnum/what-you-lose-by-ignoring-gender-a8f639764531`. Medium. (Accessed: 2020-6-7)

Elish, M. C. (2019, March). Moral crumple zones: Cautionary tales in Human-Robot interaction. *Engaging Science, Technology, and Society*, *5*(0), 40–60.

Endsley, M. (n.d.). *bsdiff*.

Ensmenger, N. L. (2012). *The computer boys take over: Computers, programmers, and the politics of technical expertise*. MIT Press.

Eubanks, V. (2018). *Automating inequality: How High-Tech tools profile, police, and punish the poor*. St. Martin's Press.

Fernandes da Costa, L. (2017, February). *Understanding dependency management in go.* `https://lucasfcosta.com/2017/02/07/Understanding-Go-Dependency-Management.html`. (Accessed: 2020-5-8)

Fitch, M. (2008, February). *Venting my frustrations with PC game-dev.* `https://forum.quartertothree.com/t/venting-my-frustrations-with-pc-game-dev/42627`. (Accessed: 2020-5-24)

Fleck, L. (2012). *Genesis and development of a scientific fact*. University of Chicago

Press.
Gabriella Coleman, E. (2012). *Coding freedom: The ethics and aesthetics of hacking.* Princeton University Press.
Haraway, D. (1988). Situated knowledges: The science question in feminism and the privilege of partial perspective. *Fem. Stud.*, *14*(3), 575–599.
Haraway, D. J. (2016). *Staying with the trouble: Making kin in the chthulucene.* Duke University Press.
Harding, S. (1992). Rethinking standpoint epistemology: What is“ strong objectivity?”. *Centen. Rev.*, *36*(3), 437–470.
Harraway, D. (1997). *Modest_Witness@Second_Millennium.FemaleMan©_Meets_OncoMouse™.* Routledge New York.
Huang, A. (2003). Keeping secrets in hardware: The microsoft XboxTM case study. In *Cryptographic hardware and embedded systems - CHES 2002* (pp. 213–227). Springer Berlin Heidelberg.
Humphreys, L. (2018). *The qualified self: Social media and the accounting of everyday life.* MIT Press.
Jurgenson, N. (2019). *The social photo: On photography and social media.* Verso Books.

Kaplan, C. S. (n.d.). *Understanding the halting problem.* `http://www.cgl.uwaterloo.ca/csk/halt/`. (Accessed: 2020-6-6)
Kelty, C. M. (2008). *Two bits: The cultural significance of free software.* Duke University Press.
Kitchin, R., & Dodge, M. (2011). *Code/space: Software and everyday life.* MIT Press.
König, D. (2019a, December). *End of python-2.7 support in sahana eden.*
König, D. (2019b, April). *Fix validator_caller: run .validate() if overloaded, not if default.*
König, D. (2020, January). *Bug fix IS_PHONE_NUMBER: don't override default.*
König, D., & Drexler, D. (2020, January). *Bug fix IS_PHONE_NUMBER validator error messages.* `https://github.com/sahana/eden/pull/1531`. (Accessed: 2020-5-30)
Latour, B., & Centre de Sociologie de L'Innovation Bruno Latour. (1999). *Pandora's hope: Essays on the reality of science studies.* Harvard University Press.
Lee, A. (2020, April). Wanted urgently: People who know a half century-old computer language so states can process unemployment claims. *CNN*.
Le Guin, U. (2014, November). *Medal for distinguished contribution to american letters acceptance speech.* National Book Awards.
Mackenzie, A. (2006). *Cutting code: Software and sociality* (S. Jones, Ed.). Peter Lang Publishing.
MDN Contributors. (2020, May). *HTTP caching.* `https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching`. (Accessed: 2020-6-5)
Python Software Foundation. (2019, December). *Python release python 3.6.10.* `https://www.python.org/downloads/release/python-3610/`. (Accessed: 2020-5-10)
Reardon, J. (2017). *The postgenomic condition: Ethics, justice, and knowledge after the genome.* University of Chicago Press.
Sahana Foundation. (n.d.-a). *Making chaos managable.*

Sahana Foundation. (n.d.-b). *Technical-Overview.* `http://write.flossmanuals.net/sahana-eden/technical-overview/`. (Accessed: 2020-6-6)

Sahana Foundation. (2011, December). *Sahana eden brochure.* Online.

Sahana Foundation. (2015). *InstallationGuidelines/Windows.* `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Windows`. (Accessed: 2020-5-6)

Sahana Foundation. (2018, February). *InstallationGuidelines/Linux/Server/CherokeePostgreSQL.* `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Linux/Server/CherokeePostgreSQL`. (Accessed: 2020-5-23)

Said, E. W. (1979). *Orientalism*. Vintage Books.

Schüll, N. D. (2012). *Addiction by design: Machine gambling in las vegas*. Princeton University Press.

Sismondo, S. (2008). Science and technology studies and an engaged program. *The handbook of science and technology studies*, *3*, 13–32.

Star, S. L., & Griesemer, J. R. (1989). Institutional ecology, translations' and boundary objects: Amateurs and professionals in berkeley's museum of vertebrate zoology, 1907-39. *Soc. Stud. Sci.*, *19*(3), 387–420.

Subramaniam, B. (2014). *Ghost stories for darwin: The science of variation and the politics of diversity*. University of Illinois Press.

Suthivarakom, G. (2020, February). *Welcome to the era of fake products.* `https://thewirecutter.com/blog/amazon-counterfeit-fake-products/`. Wirecutter. (Accessed: 2020-5-24)

The Python Software Foundation. (2020, May). *Glossary.* `https://docs.python.org/3.6/glossary.html`. (Accessed: 2020-5-22)

Traweek, S. (2009). *Beamtimes and lifetimes*. Harvard University Press.

trendspotter, König, D., & Boon, F. (2020, April). *Several various NoneType errors (setup, vol, project, inv, dc, ...).* `https://github.com/sahana/eden/issues/1543`. (Accessed: 2020-5-24)

Ullman, E. (2012). *Close to the machine: Technophilia and its discontents*. Picador.

Yuill, S. (2008). INTERRUPT. In M. Fuller (Ed.), *Software studies: A lexicon* (pp. 161–168). MIT Press.

Zuiderent-Jerak, T. (2015). *Situated intervention: Sociological experiments in health care*. MIT Press.