Software as Composites: A Maze of Twisty Passages

Daniel "Drex" Drexler

Center for Science, Technology and Society at Drexel University

Author Note

Thank you to my project advisor Dr. Vincent Duclos and the rest of my committee: Dr. Gwen Ottinger, and Dr. Alison Kenner. Particular thanks to Dr. Ottinger who spent an enormous amount of her time helping me develop my ideas. This project would not exist without her help. Thanks to the entire Center for Science, Technology & Society at Drexel university. I owe everything I learned to the many kind, patient and brilliant faculty with whom I had the privilege to work.

Abstract

Software presents new problems for understanding how creator agency is expressed through the material character of tools. Existing work emphasizes both the relationships at the center of software creation and the tendency for personal bias to drive the behavior of tools. I investigate how these forces are mediated by the material qualities of software, especially the software creation process, and how they appear in the behavior of assembled pieces of software in the world. I find that existing theoretical models that might be used to understand how software impacts the world are insufficiently reflective of the degree to which those who work on software are unable to predict its behavior. In response to my investigation and my own work in software, I suggest a new way of thinking about software, composites, whose qualities encompass both the tendency of software to faithfully transmit the intent of its creators and the potential for it to subvert that intent.

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 can be explained by the hidden or 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. The relationship between past and present software artifacts . Relationship between humans working on the project. Relationship to the imagined future. Software also exists within ecosystems: code written for a particular software project is surrounded and supported by code which is used by many software projects. 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 appears to be what Latour called a "blackbox" - a container that hides complexity. But we will see that that idea of a blackbox obscures important qualities of software. 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 artifact components, in contrast, often overlap and interleave. This level of complexity lead software artifact to be predictably unpredictable, a tendency which developers have responded to with surveillance practices. I think understanding software artifact as composites - fixed heterogeneous arrangements of components. A composites' behavior emerges out of the behavior of individual components, mediated by the particular arrangement of those components. Software artifact are not changed post-assembly, so while users always encounter them assembled, developers are always

thinking about and changing with pre-assembly components. Developers leave imprints of their views on each component, and ensure that those views survive assembly with surveillance-based testing practices. However, due the limits of developer attention, unforeseen qualities of composites constantly threaten to emerge unexpectedly when users interact with them. Because composites often have other composites as components, these unforeseen qualities can emanate from any element. Their unpredictable nature is what sets composites apart from other tools.

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) are also 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. A particular software artifact cannot be usefully separated from the ongoing social, economic, cultural and organizational context in which that software artifact. Software is designed by the people who make it to fit within their view of the world and, simultaneously, to perform the material role that software might usefully fill (Mackenzie, 2006). Software is both material and socially defined.

In order for a future software artifact to have certain properties, the people making that software artifact it must first imagine those properties in a shared social context. Like Fleck (2012)'s thought collective, people need to imagine futures together. The material work of preparing software to deal with a world is always predicated on needing to imagine the world that software will find itself in. This both leads to collective planning between software projects and also means that the material qualities of software artifacts reflect the qualities that their creators believed would be useful in the future. Examining the details of what software does now will tell us about what its creators imagined the would might be.

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 experienced a back-and-forth between what was technically possible and socially desirable. They eventually settled on a solution that allowed lacuna to exist in the database in return for a more 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. That ecosystem is at the end of a chain of decisions to build one capability over another - this 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. Social systems demand material changes in software so that the software can properly reflect the social reality.

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, our individual social horizons have been so dramatically expanded by networked communication that many more people experience a much greater diversity of social perception than ever before (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 the bottle is how we get to the wine. It matters. 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 qualities of software materially 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 that the material character of technology and the social goals of its builders are necessarily connected. 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 through describing the impact of those qualities on the social world.

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]. Code does not have agency, but its execution (like the performance of a speech) has an efficacy that exceeds its written form. Reading a program is one thing, experiencing its execution is another (Cox & McLean, 2013).

There are even pre-computer arguments for seeing computer software as having unusual and deeply important material realities. Hacking, a cultural movement that began before software, is focused on joyful play that elides and subverts objects' intended purpose (D. Drexler, 2019; Gabriella Coleman, 2012). It's no accident that hacking has become so associated with digital technology. There is no other medium better suited to both playfulness and immediate access to material impact on the world.

Materialized Perspectives

A model is worked, and it does work

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

My account of the pluripotency of software and its particular material impacts begs a question about software: how do we account for the difference between what its makers intend and what software does? Software is epistemic - it functions through selecting and acting on representations 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 nearly accidentally (syphilis retains a moral dimension to this day) or it can be central to how a field of thought operates (like the defunct field of orientalism) (Fleck, 2012; Said, 1979).

[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 set apart (even in an pejorative way) helped them imagine different understandings of intellectual property. 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).

The theoretical framework I'm using is Harding (1992) and D. Haraway (1988)'s Standpoint Theory. Standpoint Theory connects the background of people making epistemic decisions to the epistemic representations they select. Without those connections, ideas become what D. Haraway (1988) calls a god trick: the appearance of knowledge simply *being*, as if it came from nowhere, creating the appearance of objectivity. Truths are found first by people most able to see them. Connecting the circumstances that led them to that discovery strengthens our understanding of that truth. God tricks can also be found in tools. Choosing one way over another without situating (explaining) that choice creates the appearance there is no other possible choice. This allows problematic understandings, like a eugenic understanding of population, to remain within tools without critique (Subramaniam, 2014). God tricks also aid interested parties in portraying limited views of complex systems as the ultimate goal. The focus on genes in genetic research was enormously profitable for some, but was not enough to effectuate revolutionary change in the enormously complex system of biological life. Harraway (1997); Reardon (2017). Problematic characters of systems are allowed to stay hidden because the problematic qualities are not problematic for those who own the systems.

It is not necessary for a system to become problematic before we can learn by investigating 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 involve more 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) found that philosophical differences between teams of high energy physicists led them to make dramatically different particle detectors, her findings did not need to find a problem to be of use. This project is interested in finding choices that software makes but does not acknowledge. God tricks escape from documentation or acknowledgment. They are useful markers for further investigation and may be useful in explaining how

software came to be as we experience it.

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).

Better understanding the nature of software isolated from social situations will enable more complex work that studies both. I'm interested in trying to engage directly with the world Ullman (2012, p. 3) says you find when you enter the world of computers and "pass through a membrane where the real world [no longer matters."

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 a central question: what *are* software artifacts? How do understand these things? How do they exist in the world? 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' languages[2] (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. Software artifact live in the same world as we do, even if their way of existing is very different. Understanding how they impact us demands we understand how they exist.

I find Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s blackboxes a good basis[3] for thinking about software artifacts and how they 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 artifacts, 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. Once a blackbox failed, the fact that it has components becomes socially available and socially unavoidable. What is inside a blackbox varies, but they often contain other blackboxes. Like blackboxes, software artifacts are complicated objects that move around the world with socially invisible internals.

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)*

[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).

[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).

So in one hand I hold Latour's blackboxes and in the other Harding (1992) and D. Haraway (1988)'s Standpoint Theory. As Mackenzie (2006) recounts, the process that produces software is complex, relational, 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?

The first step to to try doing real work on real software. Social scientists can and must take sides on the issues they study and intervene to gain firsthand knowledge on the subjects we claim be experts on. Medical professionals use interventions (and their outcomes) as tools to generate knowledge and test assumptions (Zuiderent-Jerak, 2015). My own pre-academic experience as a software engineer supports the knowledge generating power of careful and measured intervention. I will be looking for materialized perspectives to investigate within my site and, though engaging with the source code and the software artifacts of that site, learn something about how we might address materialized bias in situations where the stakes are higher.

The object of study

Software in General

Before diving in, I should explain some ideas I am going to be using frequently. Some of these are rooted in the literature I just went over and some are simply particular understandings of common ideas.

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 carry out the instructions encoded within the artifact ('execute' it). Each component is equally a part of the artifact, but some components are given power over others in the assembly process, leading some components to have more influence than others. Software is largely distributed in the form of artifacts, though development generally takes place on individual components.

A **software project** is, as Mackenzie (2006) says, the extended social, material and organizational structure that produces "software." These projects are not unified by the production of software artifacts per-se. They can also be focused on creating standards that allow software artifacts to interact, or managing the way that software warps space (Kitchin & Dodge, 2011). 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 people who are part of a software project are acting both as themselves and also as agents of the larger undertaking. When people are acting as agents of a larger project they will certainly have their own thoughts an opinions about any particular action, but the body of work they contributing to is more the result of the social context inside the project than it is the particular person who happens to be acting.

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).

Method

This project was designed to take advantage of our ability to learn about things by changing them. It's a practice I'm familiar with from my personal background in software engineering, but it's a practice that many other disciplines use (Zuiderent-Jerak, 2015).

My plan was to engage with my site (Eden) as both a programmer and a user. I would learn its background, learn how to set it up, learn how it works, learn how it might fail. This happened on multiple levels. First on the level where I interact with software projects and software artifacts as blackboxes, without looking inside to understand how they work. Second, going back and investigating what was happening on the inside (and disrupting the idea that software artifact are blackboxes). Third, writing my own small software project that produces its own software artifact in reaction to what I found in my investigations. The mix of connecting theory to existing work and writing my own response to those theoretical connections will, I hope, yield novel results.

Because of my limited time and resources, this my engagement is not ethnographic[4]. I hope the experiment of applying social theory to software outside of the social context that software was created in is useful. I hope that whatever I learn can be applied to software in its full social context, but I could be wrong.

---

[4] As I explain later, I had some contact with the Eden team that was influential on my thinking. I hope I draw conclusions from my own understanding of their words and I avoid characterizing the culture inside Eden.

What is EDEN?

That virtual worlds are places means they can be fieldsites;

Boellstorff (2015, p. 107)

The particular piece of software I engaged with is Emergency Development ENvironment (Eden). Eden was originally created by a coalition of Sri Lankan Information and Communications Technology (ICT) companies in response to 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 includes 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 a single central authority. The system tries 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. 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 is also designed 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. The range and breadth of its functions are impressive and seem to reflect Eden's repeated use as a response to disasters.

Why EDEN. I selected Eden for this project for two reasons: it is not obviously problematic and it is a Free open source software (FOSS) software project in a language I am familiar with. The idea of improving software is difficult and dependent on the question "for whom"? 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.

I chose these limits, in part, because the the final project for a masters degree is relatively short. I performed this work over during nine graduate units of class work. I didn't feel like I had time to find *and* fix flaws. For similar reasons, I selected a project that I would not have to apply to gain access to its source code, and also one written in a language I was already familiar with[5].

Finally, though project does not investigate the social world of Eden, it also does not imagine that technical qualities exist independently from social ones. 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

---

[5] Even so, I managed to make several serious errors in the software I wrote for this project.

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 recommended 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). FOSS is both a philosophy and a system of development that has material impacts (Kelty, 2008). One of the 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 their latest source code. They also maintain full histories of who changed what, when it was changed. Often they also record why *that particular* change was made over any other possible change (Chacon & Straub, 2014, p. 13-16). Though there are many systems for this, the most popular tool is Git, and it is the tool the Eden uses.

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 compatible 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 would certainly possible if given enough time, but it did not seem possible given the constraints of my project. So, I have chosen to focus on a limited number of Python

libraries used by Eden[6].

Table 1
Purpose of libraries

Component Name	Brief Description of Functionality
Eden	Implements the logic for managing the various resources tracked by Eden
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

The Eden component I chose because it contains all the Python code written by the Eden project. It is also placed in the most powerful position inside an Eden software artifact. It is the starting point for nearly every operation within the Eden software artifact and so including it is essential.

I do not know that the other three components are the most influential components. That judgment would require a good deal more study of Eden than I have done. I would also need to develop a method to compare functionality across programming language boundaries. For the same reason, my analysis omits all non-Python components (databases, etc). This is another omission caused by time limits as opposed to a considered opinion that those components contribute less to the behavior of Eden.

I focus on the components in Table 1 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). That each component includes prominent references in their documentation to the other components suggests this is a suitable set to focus on.

---

[6] 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 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.

It's foreseeable, but unlikely, that my conclusions might not extend to the entire software artifact / project. While it is likely that the Eden component is the most important part of the Eden, small changes elsewhere can have large impacts. I will show how much of the code that impacts elements within one Component resides in another component. It may be useful to refer to the size (in lines of code) of the various libraries to get a sense of their relative size.

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 I imagine such exceptional circumstances would be evenly distributed. 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.

Chronicling my Exploration of EDEN

Now I want to talk about some of what I saw when I was investigating the Eden software project. I'll go over my experience of trying to assemble a working Eden software artifact, a brief and revealing conversation with developers, and finally talk about the software I wrote to try and respond to what I saw.

What is a Validator?

Though I looked at a lot of the capabilities of Eden software artifacts, both of my detailed anecdotes involved a single element of Eden: Validators. Validators are Python objects that perform a small but essential role that helps humans reliably interact with machines. Many operations within software artifacts will only work on certain kinds of data. If you ask a human to mathematically multiply one word by another word they will just be confused. If you ask a software artifact to do the same thing, it will generally cease functioning. Ensuring that data inside a software artifact is of the expected format is important. Along with the core purpose of checking the format of data, Validators do a great deal of housekeeping: informing the user of problems, passing correct data to the next part of the software artifact, etc.

Each Validator is a Python object that ensures that any data it is given fits a particular format. I will focus most on one that checks the format of phone numbers, but some others checks for latitude, or if the data is a UTC date & time. Not all of the Python code that defines the behavior of these objects exists in the Eden source code. The original Validator class exists in the web2py source code. The code in web2py defines how all Validators behave in common situations. A coding practice called inheritance allows the creating new Python objects which inherit all the qualities of another Python object, with the ability to override any aspect of that behavior. The web2py Validator object is designed to be inherited from. The section of the web2py Validator that approves or rejects a particular piece of data is isolated, so other software project can override the core function and keep all the housekeeping. The design approach of transmitting behavior without forcing the developer to remake it is very

Latour's blackboxes[7].

The Eden Validators inherit from the web2py Validator. This means that the Eden source code only needs to write source code that approves or rejects data. This is *both* a social and material quality. There is a social division where certain aspects are provided by the web2py project and certain aspects are provided by the Eden project. There is also a material reality where, when Python assembles an Eden software artifact, the Eden source code and web2py are combined into a single Python object.

Assembling a Software Artifact

> *If you wish to make an apple pie from scratch, you must first invent the universe.*
>
> *Sagan (2010)*

My work began, as all software work does, with collecting the components needed to assemble an Eden software artifact. The necessary components are heterogeneous. They include Python libraries, a Database software artifact, a web-server software artifact, a software artifact that manages communication between the Python software artifact and the web-server. It is a large and complex ecosystem. Some of the components are Python source code, some require their own assembly process to create the needed software artifact, some are distributed as software artifacts. One can learn a lot by just reading source code and design documents and descriptions of how Eden is expected to work, but there's no replacement for the process of seeing how the various elements fit together.

The Eden project has a few different resources that help developers get a copy of an Eden software artifact. They provide a Virtual Machine which will install a Eden in its own environment. They also publish a number of scripts that, when executed, create a Eden software artifact. Each script is differentiated through the primary components

[7] Within computer science, abstraction is a key concept. Components present an interface that will produce certain outputs for certain inputs. Developers are encouraged to treat this as a blackbox as, unless they need to understand the internal details of the component they are using, understanding it is seen as unlikely to be useful.

it uses and / or the digital environment it is designed to operate in (Sahana Foundation, 2018).

All of the scripts and guides published by the Eden project recommend components from a particular time period, but also recommend using the latest version of the Eden source code. In particular, Eden recommends using components from around 2014 (Canonical, 2020; Sahana Foundation, 2018). Many of the software projects that make those components no longer support the versions from 2014 and therefore say no one should use them[8]. Some users have written their own scripts with newer components and shared them on Eden the mailing list, but I could not find any official scripts or guides that offered updated components.

I thought this focus on an older set of components was interesting. Officially, Eden supports a range of components released between 2010 and 2020. They have also mentioned that they are ending support for older versions of Python (König, 2019a). Still, none of their official guides describe how to use any components other than the set from 2014. Even after announcing a move from Python 2.7 to 3.5+, their official assembly instructions all use Python 2.7. In addition, they never comment on their choice to focus on this particular set of components. I thought this was especially odd given that so many of the components makers recommended they not be used. This was the first unacknowledged materialized perspective I found and one of the first places I decided to explore more deeply.

Investigating Assembly. I decided that the way I wanted to approach investigating the selection of components was to try and use a different, more recent set. My general goal was to select component versions that were still supported by their software projects. However, as soon as I embraced that goal, I encountered an unexpected problem: the Eden component list is not comprehensive.

---

[8] Software is an ongoing process not a fixed object (Mackenzie, 2006). Software project are always making decisions about how they should allocate their time and energy. This can involve changing how they structure their software artifacts, but it also generally involves maintaining some number of existing software artifact. Because software artifact are complicated, it's common for them to contain bugs the software project did not initially detect. When those bugs are found, the software project generally reacts by going back and updating their old software artifact. However, time and resources are finite, so software projects generally place limits on how long they update older releases.

When the Eden project lists what component versions they support, they do not list every single component. To do that, they would need to release a full script or guide that lists every individual library. This put me in a strange position once I selected Python 3.6.9 (released 2019) as my first component. The only official Eden resource I could find are the installation scripts with component lists for Python 2.7 (Sahana Foundation, 2018). This meant that I needed to go, component by component, checking to see if that particular component was compatible with Python 3.6.9.

It turns out, unsurprisingly, that 6 years is a long time in the world of software projects. Though some Eden components had not needed an update since the versions released in 2014, many did. In addition, some components were only compatible with Python 2.7.X and I needed to find replacements to work with 3.6.X. This problem is not new! Python has been engaged in a slow, uneven journey away from 2.7 and towards the version 3 software artifacts for over a decade. There are plentiful resources for people searching for newer versions of components that only support 2.7.

Still, the right choice isn't always clear. The assembly of Eden is a heterogeneous process: try to install some Python components, investigate problems, install newly discovered requirements or replace components that won't install. It's a mix of running scripts that are part of the Eden assembly instructions, running scripts that assemble the components Eden uses, and manual work of searching for new options when a dead end is reached.

That being said, I generally selected the most recent stable version[9] of a given component, treating them like blackboxes that should continue to behave similarly (Latour & Centre de Sociologie de L'Innovation Bruno Latour, 1999). In most cases this worked easily, but it had one interesting and revealing issue. When it came to Eden's

[9] Software projects are aware that that future will come. They know that it takes time and attention to keep a projects' relationship with all of its components current. Some users of a software project want to have access to software artifacts with the latest changes, no matter how unreliable they are. Some users of software projects are using their software artifacts for tried-and-true core features. In response software projects follow a somewhat standard method for releasing changes. Stable versions are released rarely. They often have many changes from the last stable version, but those changes have been tested and examined for a relatively long period. Often software projects will release potential stable releases as "release candidates" as a final check to validate their reliability. Unstable versions, on the other hand, are released often. They have relatively little testing, a high chance of containing bugs and often contain large changes.

most prominent non-Eden component, web2py, I selected their latest stable release (2.18.5). This was a large change from the version supported in some Eden installation scripts, but other Eden installation scripts used it[10]. I did not expect problems.

Relationship Trouble. By all appearances I had a working Eden software artifact. I could navigate around, look at all the categories of resource Eden tracks. I could log into the accounts that Eden inserts directly into the database. The problem came when I tried to add anything new to the system: Eden would crash. The method used to provide Python web services like Eden mean that, even if it crashes when you try do to one thing, another copy of the Eden software artifact will be created to keep trying to answer requests. However, my software artifact wasn't much use if I couldn't enter any information. Eden, as I said, is a system for managing resources and I wasn't able to add any resources.

So what was going wrong? It turned out I was having my first encounter with the Validators that I described in [**What is a Validator?**]. Specifically, I was seeing what a relationship failure between different software projects looked like (Mackenzie, 2006). It turns out that one of the changes that the web2py project made was changing how they invoked part of the Validator that accepts or rejects a datum. They attempted to continue to support the old way of doing it, but there was a mistake in that attempt. This broken relationship meant that the software artifact I had successfully assembled had extremely limited capabilities. All of the systems for adding data through the software artifact used the Validators, so any attempt to add data would fail[11].

The problem, it turns out, is a single line of code in web2py. The web2py team had thought they had repaired the problem before release, and a developer from the Eden project submitted a fix[12]for the problem to the web2py project a day after the release of 2.18.5. This seems like a clear example of a mistake and one that was quickly

---

[10] As we will see, they made some small modifications to it that I missed.

[11] This probably isn't totally true. Eden implements some Validators of their own, but they also use many of the Validators sub-types that web2py provides. There may be some places where, because there are no Eden Validators, data can be entered - but I did not find any.

repaired.

It's also a very interesting mistake. I accidentally created an object that crystallizes the moment that two projects fell out of relationship with one another. All web2py releases prior to and after 2.18.5 work. It's also an error that can only be seen after assembly. Both web2py and Eden work perfectly well on their own. The new method web2py created for invoking the core behavior of Validators looks good. The Validator code in Eden likewise does exactly what it's intended to do. Like a puzzle, both pieces appear well shaped and have no obvious defect until you try to combine them.

Reflecting on A Relationship Break. This left me pondering several observations that I want to raise before moving on with results.

First, whose agency is being expressed here? There is extensive sociological work on how tools affect the world in ways that agree with the creators intent (Mackenzie, 2006; Subramaniam, 2014). But that doesn't seem to be what is happening here. Neither the Eden project nor the web2py project intended to break the relationship. I wasn't trying to either. Of course - mistakes happen! This is clearly a mistake, but a mistake that had enormous material consequences on the behavior of the Eden software artifact. This problem, which can be fixed by changing a single line of code, subverted the intent of both Eden and web2py teams. I suspect that there is a more complicated relationships between agency and software artifact than in other tools.

Second, this doesn't quite fit with the idea I started with that software artifacts are blackboxes. In one sense, the Eden software artifact looks just like a blackbox: a uniform exterior with a complex interior. However, it also subverts the idea of blackboxes containing other blackboxes. Where is this problem? In the Eden component blackbox? If any other version of web2py would work, are the contents socially invisible? This seems less like an object whose complex contents can remain socially invisible than something whose identity derives from the exact combination of its components.

---

[12] The main script I was following contained a fix for this problem in a slightly more compact form. I'm embarrassed to say that I missed it, but I think it led me down an interesting path.

Finally, my experience of needing to select particular components makes me reflect on the choice by the Eden project to recommend older components. I suspect the recommend the older components because it's more predictable, less likely to have unexpected interactions like this. Of course, this whole experience is about the unexpected. The web2py team did not notice a problem that Eden noticed immediately. The problem was equally discoverable by both teams, anyone can build an Eden software artifact. What is the right way to think about the extremely large number of possible components? What does it say about software?

For now, let's move on to when I have a more-or-less a fully functional software artifact and my next encounter with Validators.

Reckoning with Failures

After assembling a fully 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 on the form that this number number should be formatted in a particular way. I didn't think anything the lack of instruction until Eden informed me that I needed to "Enter a valid phone number" (König & Drexler, 2020). I didn't understand because my phone number is valid - it's a phone number. I had followed the common conventions for phone number formats ('xxx.xxx.xxxx' as well as 'xxx-xxx-xxxx' and 'xxxxxxxxxx'). The software artifact wasn't making sense, I needed to look at the source code. When I investigated the the Eden part of the source code I discovered that I was back in the Validators. This phone number Validator had a setting that I was not aware of to require the phone number in international format (beginning with a '+' and a country code). Also, the Validator seemed to have a bug in its message code - it was saying the wrong thing (complaining that the input was not a phone number instead of complaining it was not an international phone number). Now knowing what to do, I entered the phone number in the format the Validator expected and created my profile.

This seemed like an opportunity to investigate. It appears, if one only has access to the software artifact, that there are two layers of unacknowledged materialized perspectives: requiring international format numbers without mentioning it and treating all non-international numbers as invalid. Because I have access to the source code I can tell that the Eden project intends to require international numbers and the confusing error message is an error. However, I think this problem highlights the power

of perspective and I want to expand on it a little.

A Matter Of Perspective. First, imagine if the software artifact returned the right error message. It told users their number needed to be international (instead of valid). 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. The user is confused about what the software artifact needs to allow them to advance - do they respond to the error message or the field asking for any phone number?

Second, we can imagine an Eden software artifact that asks a user to use the international format. But, if the user leaves the code off, they are told their number is invalid. Again the user would be confused about what the software artifact is seeking: does their number need to be put in the international format, or is there something that makes the number they entered "invalid"?

I can tell, by looking at the source code of the Eden Validator, the difference between returning an incorrect error message and omitting a message to the user about a requirement. But I can only tell because I can see the source code. For a user who only has access to the assembled software artifact, both situations come down to the software artifact sending different messages and the user needing to decide which message to 'trust' in order to advance. We can start to see here the difference in how users and developers, those who can see inside the software artifact and are stuck outside, interact differently with these objects.

Seeing Eden Through Another's Eyes. So, after discovering it, I decided to see what I could learn by contributing a bug fix to the Eden project and fixing this (extremely minor) bug. I wrote and submitted a small change to the Eden components source code that intended to fix the problem. Unfortunately, much to my embarrassment, my fix unintentionally changed other behavior - 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 wasn't accepted, one of the Eden project

developers said several interesting things 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)[13]

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 that testing too much could be bad. This might come from the extra work of writing tests, the ongoing work of keeping tests up to date as messages change, or standards involving all tests passing at all times[14]. In any case, the decision to limit testing is also a decision to, in practice, pay attention to a particular subset of all software artifact behavior. It's also a decision to choose not to watch some behavior of Eden software artifacts that the developers know users will see. Development would demand more attention if the full scope of possible functionality was checked. So, it's better to be ignorant about some less important behavior.

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 the Validator works if it accepts numbers in the desired format and rejects others. The message the Validator sends matters less. This seems to be a world view that, if any component is expected to have multiple outputs for a given input, each output should be considered working or

---

[13] 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 software artifact.

[14] It's here 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.

non-working separately and that there is a hierarchy to which functions are more or less important.

Taking Action

Now that I've covered a couple of highlights from my explorations of Eden, I want to talk about the software I wrote in response to what I found.

A primary motivation for this project was the sense that Standpoint Theory 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 falsities that appear true because the falsity and the person exist within a power structure that supports the view that the falsity is true (D. Haraway, 1988). Standpoint Theory explains the mechanism through which we inappropriately attribute objectivity to things that are actually compelling because of personal background, but it is of little use in understanding to affect changes to the things.

If software projects believe false things because of the collective backgrounds of their members, convincing them that those views are false does not help them address it. The same cultural limitations that hem in action would also hamper any effort to escape from those cultural limitations.

So I hoped to find ways to better address the standpoints of development teams. I thought that starting from software elements that had unacknowledged materialized perspectives would help me imagine a response in software. I hoped to give teams tools to explicate what is happening in some way. 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 view of the Eden project. If I want to improve Eden for them they have instructions. The Eden project Github has a list of things they would like to change and an open invitation for anyone to help them. The project also communicates

about their long term plans. They are open and encouraging to outsiders and users alike who might want to comment or ask for changes in future Eden software artifacts. These conversations often require a certain level of technical expertise to fully participate in, but the project still has clear ideas about what it means to improve the project. The problem of software expertise shrinking who has access to software is common to FOSS software projects, but those projects can still communicate to the population who can understand them (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[15]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 am 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[16]. However, as software artifacts grow to include artifacts from many different software projects, it seemed like there were real questions of agency involved.

If a software project I am involved with is used in ways I don't agree with, giving developers more tools to assess their involvement would be useful. If I create a tool and

---

[15] 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.

[16] More often software contains the source code of one author and an assembled software artifact from a software project that handles the translation from source code to machine language.

it's used to do harm, I should consider my role in enabling that harm. The idea that developers' responsibility stops once they've released their work to the public is wrong. Software developers have worked very hard to give control to developers, both in software and in legal systems, after developers release source code or software artifacts to the public (Gabriella Coleman, 2012; Kelty, 2008). There are few easy answers to be had, but software developers need tools to help them assess how much their work has contributed to other work.

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 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 calculable set of particular components that affect that elements' function. Those components have a concrete set of authors. I hope this is a useful starting point.

I named my 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 modules mentioned in Python source files to particular files in particular places in computer 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 goal, but it turned out that py2deps did not differentiate between using some or all of a python library. It was designed to detect if a library was involved in any way. It 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 to build my own. 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 Python objects. This is possible because programming languages are designed to be mechanistically readable and makes it easier to interact with the statements in that language. The AST functionality was originally written to allow the processing and execution of Python source code by the Python software project, 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 by the name ⟨first_file⟩.

1. Create an empty index of python files named ⟨imported_files⟩.
2. Create an empty list named ⟨read_files⟩to store file locations I have seen.
3. Create a list named ⟨files_to_scan⟩with ⟨first_file⟩in it.
4. 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 the Python file at that location.
   (c) Add this file location to ⟨read_files⟩.
      i. Create a new tracking *object* named ⟨file_info⟩.
      ii. In ⟨file_info⟩, record:
         - Any *import statements* found in ⟨AST⟩.
         - The *symbol*s[17] 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 file location that it refers to:
         A. Try to locate a file using several techniques, not all of which apply to all *imports*:
            - If the *import* is relative, navigate the file structure searching for the file it refers to.
            - Ask the Python software artifact to find the file location for the *import*.
            - Ignore *import*s that fit these descriptors:
               – *Import*s 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).
               – *Import*s which, when asking the Python software artifact to find them, returned an error.
         B. If a Git repository has a copy of the Python file I just located, exchange the location I found for one in the Git repository (so its history can later be read).
      iv. Add each file locations to ⟨files_to_scan⟩as long as they meet these requirements:
         - They were not ignored as described above.
         - A file location could be found.
         - The location is not already stored in ⟨read_files⟩.

[17] 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.

        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⟩.
5. Empty the ⟨read_files⟩list.
6. Create am empty tally of authors and change numbers named ⟨changes⟩.
7. Load the ⟨file_info⟩associated with ⟨first_file⟩from ⟨imported_files⟩.
8. 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.
9. ⟨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[18].

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.

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 evolve 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. Concerns about how to properly compare compare different instances of work are 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

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

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. In addition this tool continues to ignore all non-Python code for similar reasons.

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 science, technology and society literature over a relatively short period of time. The result is imperfect. I will briefly go over known problems.

A Brief Overview of Symbols. 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.

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[19] in. Each Python file that 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 and so on.

---

[19] 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 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*.

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*[20]. 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. I will do my best to explain, but if I fully understood what was going wrong I would have simply fixed the problems!

Some Python statements that add a symbol are simple to process. Function definition statements and object definition statements each include a single name for the symbol. However, there are instances where symbol creating statements (such as assignment operations using the '=' sign) can have complicated structures. For instance, one might write: `(a, (b, (c, d))) = function_call()`. There's an implicit structure to this statement that I can write out[21], but I was unable to write Python code that handled situations like this one. I was both hampered by the structure of the code I had already written and layers of state management I would need to handle. Handling this properly would require reference objects that stand in for the results of future Python software artifact operations, which I did not have time to write.

---

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

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

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**'s current results avoid duplicate references to the same file and I believe that few of my known errors would hamper the basic author count.

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 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 non-cyclical tree graph 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 maintain a single 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 code is a mess as a result.

The procedures for associating a particular Python file with a particular Python likely could fail less often than they do now. 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 possible for a file to import a Python module that it does not use (I have unused imports in the **materiality** source code). These can be detected, but I do not do it.

Finally, there were cases where overlapping errors that, individually, would have broken the software artifact effectively canceled each other out. 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 are 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, so here are my results for one particular file in the Eden component source code.

Validators by the Numbers

Below are the results of running **materiality** on the Python file[22]containing the source code for the Validators that I repeatedly encountered. This is the file that

contains the Eden part of the Validator functionality that is partially defined in Eden project source code and partially defined in web2py source code.

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 as input.
- 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 the [**Table**] 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 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`:

I think the most interesting thing here is how the work on the shared Validator

[22] The path inside Eden's repository is /modules/s3/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%

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[23]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 skilled 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 exploring.

[23] 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 What I've Seen

Now I want to turn to drawing out the themes in the things I've seen and the work that I've done. In some ways

Who wants a Relationship to Fail?

The initial failure I encountered where I discovered a break in the relationship between web2py and Eden has got me thinking about software artifacts and agency. On one hand, small mistakes are universal and this was certainly a small mistake. When the web2py project fixed this problem, it took only two lines of code (König, 2019b). But, because this is a failure to agree on how to transfer control from code written by the web2py project to code written by the Eden project - any of the groups involved could have repaired it. Eden Could have chosen to change their source code, I could have added a new component to my Eden software artifact that changed the relationship between Eden Validators and web2py Validators.

It seems like expressing agency in software artifacts is a narrow thing. There are many elements in each component and proper software artifact behavior doesn't move through just one layer, but often through tens or hundreds of layers. Eighty six files were involved in supporting the behavior of the Validators objects. Each relationship that exists in those files is a relationship that exists on two levels: a material level as defined by the instructions in the code and a social level that exists in the humans writing that code. In order for people to imagine how to form new relationships in the material of software, they must first imagine what that relationship will be like (Mackenzie, 2006). This work of projection and imagination exists at every level of the program. All of the potential fixes (in Eden, in web2py, etc) involve envisioning the relationship in a particular way before engaging in the material work of reforming it.

So what this problem shows is an escape of the material qualities of a software artifact from the social imagining of how it should be. The escape, in this case, materially changed the structure the assembled software artifact compared to the socially envisioned software artifact. Throughout this project I've seen the process go

from the material to the social and back to the material. The Eden project rejected my bug fix because it had a material failing - in certain circumstances it returned the wrong error. But that material failing was not demonstrated in a material fashion (assembly of a software artifact), it was agreed upon socially. Any time we consider a potential problem, we cannot help but consider it in a social context because we are social creatures, but while a software artifact is created within a social context, it is not a software artifact. It is mechanical.

Each line of source code has a material effect and a socially understood intent. Human thinking is fuzzy, but machine action is not. This can manifest in simple ways. A software artifact defying developer intent by not telling the user what is happening. It can also manifest in complex ways. Systems can express bias that developers believe is not there. Systems can be contain instructions that will lead to decisions that developers believe are not possible. Once they are assembled and sent out in the world, software artifacts will do whatever their code tells them to do, whatever their creators believe.

Software Artifacts as Blackboxes

Blackboxes were used by Latour to explain how tools become involved in our accomplishing of goals. It's too simple to say that we do something or the tool does something - the tool changes us and we change the tool (Latour & Centre de Sociologie de L'Innovation Bruno Latour, 1999). On one hand this fits with how people interact with software artifacts. Throughout this process I've used software artifacts to assemble software, write this paper, distribute my work, etc. On the other hand, blackboxes are also a model to explain how complex things make up other complex things, and that does not fit what I have seen. Software artifact is hard to see as a box containing other separate boxes.

The Eden source code does not just use the web2py, it transforms it. When the Python software artifact executes the Eden (and web2py and all other components) source code, what is placed in computer storage is a mishmash of all of the components.

Once they are incorporated into a software artifact, components mix with each other in a way that is particular to that set of components and that method of assembly. Software artifact are blackboxes, but their internal structure is far more complicated than blackbox theory can capture. That complexity is key to understanding how agency is impacted and convoluted by software artifacts

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 around the discussion itself. This was 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 how most software projects impact the world: through directly or indirect contact with software artifacts the project produces or supports (Kitchin & Dodge, 2011). When we say that software impacts our lives in direct and indirect ways, that impact flows from particular people interacting with particular software artifacts. Though software has made it possible to remove humans at many of the intersections of power, it is still experienced through a multitude of relational contacts between a representative of power (a person or a software artifact) and a particular human Cheney-Lippold (2018); Deluze (1995).

Do Software Artifacts Matter. Given all of that, I think it's interesting that the entire discussion about this bug took place without referencing the behavior of any particular software artifact. There are other situations where the Eden project ask about component selection to try and understand the source of problematic behavior in a software artifact (trendspotter, König, & Boon, 2020). The idea that software artifact may behave in ways that are deceptive to the user was also an aspect of the discussion around the bug. Why isn't it useful to talk about any software artifact in particular?

When a software artifact is under discussion, that discussion often starts by trying to find 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 particular 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. The practice of patching, or distributing updates to software artifacts, has the goal of changing an existing software artifact to be identical a newly assembled software artifact. A patch program is a different software artifact that will effectuate a different assembly process given its target software artifact and itself. Patches are generated by comparing the old software artifact to the new one and storing a list of changes to make to transform the old to the new. It has the goal of producing a software artifact that is the same as the one generated by a full re-assembly with all of that software artifact's components (Endsley, n.d.). Whether a software project replaced entire artifacts when they find problems or if they create special purpose software artifacts to transform existing software artifacts, the desired software artifact is always the software artifact that is generated by a full assembly of the latest components.

The bug was fixed by changing a component with the intent of the changed component being assembled into a new software artifact. It would be possible to, for each update, write source code with the intent of creating a software artifact to repair that particular problem. This would be similar to how physical objects are repaired - building a new piece of software to carry out each reparative step. But such a practice would be too labor intensive. The closest thing is the practice of binary patching I just described, which depends on fully generating a fresh software artifact. 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 changes the worlds of the people doing the work and the particulars of how these operations effect 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 applications to software projects and software artifacts.

The Limits of Developer Attention

König expressed the idea 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.

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 code (See Table 2). The version of Eden I have been working on has seventy two authors (excluding 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. That is more code than many contributors have added to the project. 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 this about the social context that a software project exists within. 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[24].

What About when Developers Aren't Looking? This focuses on aspects of software artifact behavior that are 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.' 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.

There's also the risk that 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.

Developers choosing to watch certain parts of their software artifacts and choosing

---

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

not to watch others creates a separation between the social and material worlds around a particular software artifacts. The material behavior that developers choose to surveil is socially visible. The source code or testing practices mean that the the developers have a relationship to the idea that their software artifact should behave a particular way (Mackenzie, 2006). However, based on these experiences, it seems clear that developers do not pay equal attention to every material element of their software artifacts. There are aspects of software artifact behavior that they can't afford to pay attention to or that they believe are not important enough to pay attention to. The choice to ignore certain behavior based on the idea that it's less important means that we should be asking "less important to whom?" (D. Haraway, 1988; Harding, 1992).

It's possible (even likely) that members of a software project are the individuals who understand that software project, but the impacts of a given software project can be felt far outside the social world of that particular project (Kitchin & Dodge, 2011). Software project run the risk of being blind to serious issues when they accept an understanding of what is inside and outside the bounds of reasonable concern, because what is concerning to us is always driven by our individual and group identities.

Beyond the implications for those outside a software project, the nature of software means this has important material implications for software projects. Each component of a software artifact will have an understanding of that components' material impact that is socially constructed by the components' software project. There will be material effects of that component that are invisible to social context within its project. The project can't warn people about the material qualities of their own work that they do not see. This means that each component in a software artifact will behave in ways that the components' makers are not aware of and cannot speak about. Because each software artifact component may itself be a software artifact (and so on and so forth) there can be hundreds of thousands of layers of socially invisible material action.

Watching the Machines. I think that the work that software projects do to observe the software artifacts they create and ensure they behave as expected is almost certainly best understood as surveillance. Surveillance studies is a large field and one in

which I have little experience. Applying theories in this field to understanding how software projects seek to prevent their creations from escaping their authority would be interesting.

What does Working Mean Here?

I want to return to talking about the various experiences of functionality and non-functionality I've encountered while trying to fix the bug[25]. 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[26]. 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

---

[25] 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.

[26] 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. 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.

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 languages and another approach with compiled languages?

No, because the apparent differences are illusions. Compiled programs can also have their contents changed mid-executions[27]as well. 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. Knowing the results of a program before it executes is something that computer science says is impossible, at least with structure used by our current computers (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 pre-execution analysis). 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 assembly[28].

Why is this important? Does it matter for sociologists when assembly stops and

---

[27] Very old programs will do this intentionally: re-writing their code in memory as they execute. In the modern context, it generally happens when a program is being broken into by a hacker. The hacker will exploit insufficiently paranoid programming practices to get the software artifact to replace its current machine code with machine code supplied by the hacker.

[28] 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.

execution begins? The end of assembly is the point where human intervention into a software artifact stops being possible. 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[29]. This is doubly important because our ability to predict what a software 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 software artifact assembled during Eden's testing process is not exactly like the final software artifact (it largely differs in the database entries that compose it). For instance, the assembly error I encountered with web2py was easily detected by tests. This adds to my sense that 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. Tests can take the form of source code that is incorporated into one or more testing software artifacts or in structured human interactions with the final software artifact before releasing it to the public. In either case the intent is likely to surveil and control the behavior of the software artifact.

Reflecting on my Intervention

Now I'm going to briefly reflect on what I found by writing software of my own. I have some ideas about where people might take this in the future and how some of the shortcomings might be addressed.

Foreseeable Improvements

I will 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,

[29] 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 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.

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 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. In short, the statistics that the current software project produce suffer from a number of foreseeable problems that bias the results in uncertain ways. Not all of these scenarios are clearly detectable - text matching can be difficult and uncertain - but an effort could be made.

Change Distance. The greatest weakness in the current form of **materiality** is the flatness with which it presents its results. The most obvious improvement would be for **materiality** to characterize the distance of the various impacts in some way. At the 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 list, but this defies the nature of software. Much of this paper and this project has focused on the power of assembly and how, in this case, placing the Eden component in the most important spot within the Eden software artifact is why the Eden component is the most influential. **materiality** does not reflect that and it should.

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 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 unused link). However, such an analysis might be deceptive. There are real limits to how much we can say about how a software artifact will behave without executing it Kaplan (n.d.). As much as the current program is a view from nowhere, I want to avoid rendering a different kind of 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 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 as it executed. It helped my surveil my own software artifact. The numbers in `[]` are line numbers in the Python file. The symbol and letter inside the `<>` symbols communicates the status of the *import*. `<!S>` *import*s are system libraries and will be ignored. *Import* statements with a `<+R>` before them are relative imports whose referent has been successfully found in my computers' file system[30].

Many different graphical depictions 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.

---

[30] For example, the import starting with [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 everything I found in the Eden source code or the Eden software artifact suggests that the impact of Eden is in line with their goal (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 this 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 possible behavior is set. 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 focused on their ability to allow complex objects to be socially invisible, but that's not what I see with software artifacts. In both [**What does Working Mean Here?**] and [**What**

About when Developers Aren't Looking?] I encountered an object whose behavior defied shared social understandings. It was not that my Eden software artifact 'broke,' but that it broke in a 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[31]when they fail, software artifacts keep their secrets. I think blackboxes aren't helping me understand the social qualities particular to software.

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 into which those components are assembled. Once assembled, a composite can be mechanistically 'read' by a machine to tell that machine which action to take. This process 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. Each component makes a contribution proportional to its size to the material qualities of the composite it is integrated into. However, that contribution is mediated by layers of assembly and can place components in relationships that the component creators did not foresee and which result in the component behaving in new ways.

The identity of a composite derives from its behavior, not its label. The different

[31] 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).

software artifacts of Eden are not similar because they share the name "eden," they are similar because they are constructed from the same components and because the Eden source code is in the position of primary importance. As Eden evolves and changes over time, its social identity will "stretch" to reflect a diversity of behavior, but that same stretching betrays that multiple objects with different qualities are sharing a single label.

Composites are immensely useful objects. They can encode complex methods for weighing alternatives that could allow nearly immediate decisions in time-sensitive situations. If the required material resource are made available to the composite, it will continue until it exhausts its resources or is stopped. The humans that start them running do not need any knowledge of their internals. This also makes composites very dangerous. They are powerful conduits for systems of control and surveillance.

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[32], to predict. Multiple components or layers of assembly can interact in unforeseeable ways[33]. Composite makers respond to this quality by implementing systems of surveillance to observe their own creations, but this system has obvious flaws. In order to imagine what behavior is worth testing, composite makers must imagine what their composites might do. Systems of surveillance must be told what to surveil and composite makers cannot forsee all ways their composite might be used. In addition, these systems of surveillance often involve new composites purpose-build to observe expected behavior of the final composite. Like all composites, these surveillance composites can be expected to behave in unpredictable ways from time to time.

---

[32] 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.

[33] See **Other flaws**.

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

Standpoint Theory was motivated by the harm caused when we separate 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 understand as desirable must agree with their world view. How the makers of a composite choose to decide to release a composite into the world is a decision mediated and constructed by the culture of the people making that composite. Even though specific tests of that composite might be defined in terms of the material inputs and outputs, there will always be a layer of social consensus around how to understand the results of those tests. Situating a composite within the culture of the people who created it will make its influences socially visible, but the composite will still *behave* the same way. Its material qualities, when executed by a machine, will be the same, even if the social layer around the composite make them inaccessible to us.

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 people from that social context will reliably approve of a portion of the behavior of that composite (Mackenzie, 2006). But that composite is not trapped within that social context. If the outputs the composite produces for a given input seems useful to other groups, it may be adopted as a component. But this adoption

cannot not take just one part of the composite. It must take the entire thing. Including any behavior its original creators missed, or behavior its new adopters have misunderstood. Composites are mechanical things masquerading as cultural.

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 have 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 effects in components and, ultimately, composite behavior. Tracking down the exact source of misbehavior may be possible, but requires a great deal of work. The Eden composite is entirely open source. I could investigate any element of any component. That is not typical. Many of the composites that are used in other composites are closed source. They are protected from modification or investigation by copyright law (Kelty, 2008). This means that flawed composites are rarely, if ever, fixed directly. Instead, a new element would be added to the assembly process to inhibit or sideline the misbehaving component. Every composite is a patchwork of immediate solutions to problems of enormous complexity[34].

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 live in 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 of composites by another common feature of our world: humans. I

---

[34] A quality they share with many socially important objects

think composites are the closest we have come to realizing the dream of creating a new kind of creature. As speculative fiction authors have pointed out the results are often nightmarish, but they can also be 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 people live in interact with each other.

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 will lead to composites suddenly, unexpectedly diverging from social expectations. 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.

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, 22

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, 58, 59, 60, 61, 62

Eden The Emergency Development ENvironment, developed by the Sahana Foundation, is an Open Source Humanitarian Platform which can be used to provide solutions for Disaster Management, Development, and Environmental Management sectors. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 37, 41, 42, 43, 44, 45, 46, 47, 48, 49, 52, 53, 54, 57, 58, 59, 61

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 contributors, 2020). The open source nature of Python means that it is likely that Python 2 will continue to be updated by users after official support ends. . 12, 18, 19, 21, 22, 23, 24, 25, 34, 35, 36, 37, 38, 39, 40, 41, 43, 45, 51, 52, 54, 56

Sahana Foundation The nonprofit that manages the EDEN project. Based in Los Angeles with an international scope and focuses on supporting communities in disaster preparedness and response through information technology.. 16

software project Drawing from Mackenzie (2006), this is the total social, material and organizational structure associated with the production, distribution and use of a particular piece of software. This includes the people who work directly on the source code of any software artifact associated with the project as well as anyone who is part of the organization that manages, funds and directs the development of the software. It may also involve people who are not employed by the formal owner of the software project (this is especially true for open source developers). Any software artifacts assembled by members of the project who are acting as members (and not on their own) are also part of the project. software artifacts assembled using source code from the project are not part of the project per-se, but the software project bears some measure of responsibility for how those software artifacts function and act in the world. 5, 7, 14, 15, 17, 21, 23, 24, 25, 32, 33, 35, 41, 47, 49, 50, 51, 53, 54, 57, 58

software artifact An entity of uncertain composition that behaves, for common users (as opposed to technical staff) as a single entity. A software-artifact could be a single binary with all its functionality provided by machine code included in that single location in memory. It could also be a binary that makes calls to various libraries. In the case of EDEN, it's an expensive set of Python files - some of which are human readable and some of which have been modified to make them faster to read. In this case a separate program, the Python binary itself, reads and executes the EDEN program. Even though EDEN is many separate files and contains no machine code, it is properly considered a software-artifact because users interact with it as a single entity. Software-artifacts are produced periodically by software projects, but are not themselves entirely "software." The idea of software must be large enough to allow multiple software-artifacts to exist within the same project: the current version of your phone OS and the next version, the Facebook app on your phone and the web server that tells it what has happened. Nevertheless, when we are using "software" we are always directly interacting with one or more software-artifact(s). They are the material reality of software projects. 5, 6, 7, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35, 36, 38, 39, 41, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 64

Virtual Machine A family of packaging techniques that allow packaging and distribution of an entire computing environment (operating system, libraries, programs) in a single digital object. This allows one physical computer to run several independent "machines" simultaneously. When a virtual machine is started, it the hardware it sees is virtual hardware simulated by the program "hosting" the virtual machine. The hosting program then translates requests to that virtual hardware into requests to the real hardware and relays the responses. Running a program in a virtual machine is slightly less efficient than running it on real hardware, but this cost is often less than the cost of manually arranging non-virtual installs on real machines. 22

web2py An open source web framework for Python 2.7 and 3+. Provides elements that handle the common tasks of providing a web service: producing HTML, handling headers, communicating with web servers, responding appropriately to various HTTP codes. A primary component of Eden. 21, 22, 25, 26, 27, 42, 44, 45

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 . 18, 19

Acronyms

Eden Emergency Development ENvironment. 16

FOSS Free open source software. 8, 9, 17, 18, 33, 42

ICT Information and Communications Technology. 16

Additional Thanks

This project was produced in a digital soup and I have drawn on a nearly endless list of resources in order to complete it. These are not sources I cited, but without them you would not have the work before you.

Latex

- http://www.personal.ceu.hu/tex/breaking.htm
- https://github.com/SublimeText/LaTeXTools/issues/1082
- https://www.overleaf.com/latex/templates/your-apa6-style-manuscript/kngbbqpypjcq
- https://gking.harvard.edu/files/natnotes2.pdf
- https://tex.stackexchange.com/questions/74170/have-new-line-between-paragraphs-no-indentation
- https://tex.stackexchange.com/questions/171803/change-font-size-of-the-verbatim-environment
- https://tex.stackexchange.com/questions/32208/footnote-runs-onto-second-page
- https://stackoverflow.com/questions/2895780/how-to-code-tables-with-multi-line-cells
- https://tex.stackexchange.com/questions/62278/problems-with-endfloat-package
- https://www.overleaf.com/learn/latex/code_listing
- https://tex.stackexchange.com/questions/171803/change-font-size-of-the-verbatim-environment

Failed attempt to incorporate images

- https://tex.stackexchange.com/questions/542766/inkscape-1-0-not-able-to-export-files-needed-for-svg-package

- https://tex.stackexchange.com/questions/2099/how-to-include-svg-diagrams-in-latex
- https://www.overleaf.com/learn/latex/Inserting_Images

Other Resources

- https://stackoverflow.com/questions/1945075/how-do-i-create-binary-patches

Python

- https://stackoverflow.com/questions/17912307/u-ufeff-in-python-string
- https://gist.github.com/thatalextaylor/7408395

Tools

- https://www.kammerl.de/ascii/AsciiSignature.php
- https://www.tablesgenerator.com/

Tables

These tables are missing some information that is available in this Google sheet: https://bit.ly/Eden_Stats. 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.

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. (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.

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.

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.* `https://github.com/web2py/pydal/commit/cecd77127c122404c1aee7f6377c6a0150d86d84`.

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.

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.

Reardon, J. (2017). *The postgenomic condition: Ethics, justice, and knowledge after the genome*. University of Chicago Press.

Sagan, C. (2010, January). *If you wish to make an apple pie from scratch, you must first invent the universe (clip).* Cosmos. Youtube.

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.

Statz, P. (2010, February). Get started with web frameworks. *Wired*.

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

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.