Research Software Systems: Exploration and Infrastructure in Observational Cosmology

William Sutherland-Keller

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Charlotte Lee, Chair

David Ribes, Chair

James Howison

Anissa Tanweer

Program Authorized to Offer Degree:

Human Centered Design and Engineering

University of Washington

Abstract

Research Software Systems: Exploration and
Infrastructure in Observational Cosmology

William Sutherland-Keller

Chairs of the Supervisory Committee:

Charlotte Lee

Human Centered Design and Engineering

David Ribes

Human Centered Design and Engineering

In the last 20 years software has become an increasingly problematic object in the sciences. While it has become integral to research in many fields, scientific communities have struggled with problems of reproducibility, reuse, and the validity of findings produced by complex software systems. This has been attributed in part to the unplannable and unpredictable nature of scientific work itself, which makes it difficult to test research software or to define its requirements ahead of time. In this dissertation I draw on ethnographic work with a research group in the field of reionization cosmology in order to develop an understanding of how researchers develop novel software instrumentation in these contexts of uncertainty. I elaborate the notion of exploratory programming as a central process in scientific work that is distinct from software production, and I further develop the concept of the research software system as a system of heterogeneous software processes that is organized to reconcile the need for rigidity and flexibility in research infrastructures. This contribution reconceptualizes how software is used in scientific work as well as the process by which software engineering methods might be integrated into that work, and it provides new conceptual handles with which to support both activities.

Table of Contents

Chapter 1: Introduction **2**
Chapter 2: Background **10**
2.1 Scientific Software 10
2.2 Software and problems of materiality 17
2.3 Anomalies and unplannability 29
2.4 Research infrastructure and infrastructural change 33
2.5 Problems and problem formation 45
Chapter 3: Methods and methodology **51**
3.1 Methodology 51
3.2 Site Description 57
3.3 Data collection and analysis 67
3.4 Terms 74
Chapter 4: Exploration and the development of problems **76**
4.1 The generative reproduction of anomalies 78
4.2 Exploratory testing 91
4.3 Iteration and the indefinite rhythm 106
4.4 Some commitments of an examination of exploratory programming 110
Chapter 5: Software production and the research software system **116**
5.1 Production code, flexibility, and the wild west 118
5.2 Repositories and Border Markers 129
5.3 Robust code, robust phenomena 143
5.4 Research software systems 153
Chapter 6: Code growth and the development of novel capacities **157**
6.1 Code Growth 158
6.2 Graduating Code 164
6.3 The development of capacities 170
Chapter 7: Reorganizing rigor in software practice **177**
7.1 Pride and self-discipline in software practice 178
7.2 Championing 185
7.3 Champions and exemplars, human and nonhuman 191
Chapter 8: From scientific software to research software systems **198**
8.1 Software process in and of the Radio Group 200
8.2 Unplannability and the research software system 202
8.3 Sociotechnical change in software practice 209
8.4 Reapproaching scientific software 214
8.5 Conclusion and limitations 221
References **225**
Appendix **247**
Appendix A: Codebook 247
Appendix B: Glossary of terms 251

Chapter 1:
Introduction

In many fields software, and software development, are becoming integral to the day-to-day work of research (Carver et al., 2022), but scientific communities have struggled to turn software into an effective tool for scientific inquiry, wrestling with problems of reproducing findings (Stodden, 2018), building off of each other's work (Goble et al., 2011; Feinberg et al., 2020), and ascertaining the validity of results (Kelly and Sanders, 2008; Howison and Herbsleb, 2010). These problems are such that multiple programs have emerged with the goal of improving the quality of software in the sciences. These include the software carpentries, which are workshops focused on training researchers in software-related tools and techniques (The Carpentries, 2021; Wilson, 2006a); the promotion of research software engineering as a career path and a kind of work within the sciences (Baxter et al., 2012); and the creation of institutes dedicated to promoting software quality in the sciences (e.g. URSSI, 2021).

Long-running scholarship on scientific software has developed a couple of different reasons for why it has been so difficult to render software as an effective research infrastructure. Some explanations point to aspects of scientific culture, such as reward structures that favor discovery of new things over the maintenance and design of systems or aspects of training. Other explanations, however, point to a characteristic of *unplannability* in research work itself. Unplannability does not mean that researchers cannot plan at all, but rather that they often do not have specifications or gold standards to define what the output of their instrumentation should be. In software engineering terms, scientists have an "oracle problem" (Kanewala and Bieman, 2014; Kelly and Sanders, 2008). Because of this characteristic, they find it difficult to effectively test their software, or to engage in long-range planning and design work. This has led to an image of scientific software work as an *ad hoc,* unsystematic, or amethodical affair (Rother

et al., 2012; Ahalt et al., 2014). Others have argued that scientific software development resembles a nonstandard form of Agile software development (Easterbrook and Johns, 2009; Heaton and Carver, 2015).

Science studies has also long considered the problem of unplannability in the sciences, and it is something of a trope in the characterization of scientific work. This has been stated as a characteristic of experimentation (Fleck, 1979), a reason for the autonomy of science from governance (Polanyi, 1945), and an argument against efficiency metrics for scientific work (Peterson and Panofsky, 2021). In science studies too this quality of unplannability has historically been taken as part of the furniture: the unpredictability of science is clear, but there is little to be done or understood about it. Dynamics of uncertainty, contingency, and serendipity have been viewed as intractable to rationalization as aspects of a context of discovery (Popper, 1961), and have been sidelined to activities of justification or demonstration. Somewhat more recent work on science as practice (Pickering, 2010), on experimental systems (Rheinberger, 1992), and on the formation of problems in scientific work (Fujimura, 1987; Passi and Sengers, 2020) can provide a more solid basis for looking more directly at the processes by which scientists proceed without requirements or oracles.

Unplannability can also be framed as a problem that is characteristic of research infrastructures in general. Building out robust standards and systems for scientific work often runs into the problem that scientific objects change in ways not predictable ahead of time (Hirsch et al., 2022). Studies of cyberinfrastructure have looked at the processes by which researchers work through these tensions, repurposing old techniques, adding new ones, and building commensurability between old and new endeavors (Ribes and Polk, 2015). These processes are of course not just about rendering specific computing systems interoperable, but of "relational maintenance" of the interconnections between knowledge, technologies, and organizations (Bietz et al., 2012).

Taking these prior approaches together presents a number of changes in focus in understanding research software. There has been a great deal of work on how to improve or encourage the production of better quality software in the sciences, but it is also possible to consider how software becomes a productive and manageable material for *epistemic work*. Specifically, we can look at how software plays into the exploratory and contingent work that goes on in science before the epistemic dust has settled and requirements are solidly in hand. In part this means looking at the interplay between the development of software and the development of new scientific understandings. Additionally, with the right case, we can look empirically at how software engineering techniques and the best practices of research software development (Wilson, 2006a; Wilson, 2014) do come to be integrated with these exploratory or highly contingent practices. We can examine what relationships and interactions are established between exploratory research and the work of software production in the lab.

The goal of this study is to examine these dynamics of unplannability and their relationship with the development of robust research software infrastructure. I do this through ethnographic work with a research lab in the field of reionization cosmology. This group, which I will call the Radio Group, is hunting for a particular signal emitted during the early period of the development of the universe, using data collected from radio interferometers. Writing code and developing software tools is an essential part of this work and in recent years members of the Radio Group have sought to improve the way they develop and maintain their software tools. Through qualitative analysis of field notes, interviews, and documentary data I examine how the Radio Group leverages software in their research work and how in recent years they have developed processes for more robust development of research software.

In order to better understand this problem of unplannability my first research question focuses on software in a process of exploration: (RQ1) *How does software become a tool for exploratory work?* More specifically, I look at how the Radio Group leverages software in the process of

problem formation. As has been argued elsewhere, problems are the object and outcome of work in their own right (Fujimura, 1987; Leonardi, 2012). To put this differently, by the time the problem has been set a lot of work has already been done in understanding a situation and one's stance towards it. By examining this 'figuring out' work we can look at scientific work, and scientific software, in those situations of exploration where movements are made without oracles or requirements. Moreover, the work of problem formation goes a long way towards defining next steps, and, in the case of software, establishing requirements. This allows a better view of how requirements do emerge in the research process, and goes some way towards substantiating why agile-like methods work (Easterbrook and Johns, 2009).

Once we have looked more closely at processes of exploration and problem formation, we can return to the problem of introducing software engineering practices into the sciences from a new starting point. In order to do this I ask (RQ2) *How do researchers reconcile flexibility and rigidity in their software tools?* Flexibility and rigidity are broad concepts, but I look specifically at how the Radio Group maintains software as research problems and stakeholders change. I look at emergent relationships between this effort to build research software infrastructure and the more exploratory processes of research work. I examine how the work of research and the work of software production are rendered distinct processes within the research group and their collaboration, and the points of connection between these distinct processes.

Lastly, I examine processes of change that the Radio Group and their collaborators undergo in attempting to improve the way they work with software. I ask (RQ3) How does a research group incorporate software planning and design processes into their research work? The literature has framed unplannability as an obstacle to software engineering practices that require the long-term planning and design of software, but given the deep integration of such practices amongst the Radio Group and their collaborators we can ask empirically how this might be done. In particular we can examine how software engineering practices are integrated into research

practices typically marked by uncertainty and contingency. This process has thus far been taken implicitly as an individual adoption process, either through the researcher reading lists of best practices or through a software carpentry setting (Wilson, 2006a; Sutherland et al., 2025). More explicit attention to the social and collaborative dynamics around the actual uptake of these practices is much needed.

My primary approach to addressing these questions is to develop the concept of the research software system. A research software system is a system that produces novel capacities for the experience and interpretation of phenomena. One of the primary contributions of this concept is that it is an alternative to considering research software as a homogeneous category. The research software system comprises both processes of exploratory programming as well as processes of software production. These processes are distinct but mutually constitutive, meaning that the system is working well not when all of its activities have been converted to engineering or production work, but rather through the articulation and coordination of these distinct processes. It also entails different kinds of software, from research code to collaboration code or production code.

The research software system develops through iterative processes of improvisation (Weick, 1998), by which researchers draw on established routines and stable resources in order to produce novelty in the working of their instrumental system. In this “pattern-in-variety” (Cohen, 2007) view, routines are performed again each time, and they are subject to reflection and variation that can produce change in organized activity (Pentland and Rueter, 1994). Conversely, improvisations of this kind are not performed as a wild, random flailing, but rather draw on variation in established routines and artifacts as resources (Weick, 1998). Robust software is such a resource in these activities. It is in this view that we can move past the ostensible unplannability of research work. It allows us to consider how researchers pursue novelty but nevertheless structure their work and produce recognizable patterns of activity. In

this tension between established routines and the production of novelty, the research software system becomes a generative system. It becomes productive of new phenomena, new research problems, new scientific findings, new requirements, and new software.

In developing the concept of the research software system I draw on a couple of different conceptual resources. A great deal of this notion is built around Hans-Jörg Rheinberger's concept of the experimental system (Rheinberger, 1992), which provides a way of talking about the process of *differential reproduction* at the heart of systems for scientific inquiry. However, my concern diverges from Rheinberger's somewhat in that I am focused on epistemic *work*, the activities by which researchers put experimental systems into action and refine them through successive performances of the production of phenomena. This process I characterize as a transactional research process. By transactional here I draw on Deweyan (2023 [1938]) understanding that knowledge is produced through ordinary transactions with the world rather than through an external mirroring or spectation on the world (Schön, 1992). The research software system, then, proceeds through a kind of conversation with the situation, which relies heavily on action and retrospection.

A last point is that the research software system is a historical entity. Above I have described a definite system with distinct parts that interoperate, but this is not a timeless model of the way knowledge must be produced, with software or otherwise. My characterization of the research software system is also about the way specific researchers think about their work, the way they come to understand a distinction between research code and production code, or the way they come to understand notions of rigor or flexibility with software. It is also about how they enact these things. A large part of the contribution, then, is to describe how researchers articulate (Strauss, 1988) these different kinds of work, and how material artifacts are leveraged in negotiating and bounding these different activities (Lee, 2007). The research software system is itself an outcome of these coordinative activities. It is these coordinative activities that make the

research software system, and they are oriented around a tension between exploration and stability that is at the heart of the ongoing problematic of research infrastructures.

These notions of distinct processes have not always been the way that researchers think about their work, and of course it has not always been the way they pursue it. It is a way of working which has emerged in the uptake and growth of large systems of software for scientific work and with an effort to tame or systematize work around those systems. What I am describing is not just an understanding or mental model that researchers have; it is also a way of working that they pursue in the world and which more or less works for their aims. But I am not describing a fundamental law of epistemic process or a kind of social physics. The research software system is interesting as an explanation of how research work gets done with large software systems, but it is also interesting as a diagnosis of a contemporary, emerging way of working in the sciences.

I present two primary sensitizing concepts (Blumer, 1986) that can help us reapproach scientific software in a new way. The notion of exploratory programming has been described already (Kery and Myers, 2017) but I elaborate it in terms of the work practices of researchers, connecting it with temporalities of research work in the lab as well as the rituals of lab meetings and data analysis sessions. This concept of exploration on its own presents a somewhat different criterion for how to improve research work with software and what good software work looks like in the sciences. The larger sensitizing concept I develop, the research software system itself, can focus our attention on the importance of articulation work (Strauss, 1988) in making a research software system go well, particularly in highlighting transition points between different kinds of work with software, such as in the process of graduating code. These concepts not only provide new ways of approaching software as a tool in the sciences, but also in many contexts where software is frequently used as a tool in a sensemaking process, rather than as a

means towards already established ends. Such cases range from the use of programming in art and design to many contexts that go under the term data science.

The rest of this document works through these ideas in the following way. Chapters 2 and 3 present some conceptual background for my arguments and lay out my methodology and methods. Chapter 4 characterizes a process of exploratory research work both in terms of its minute activities of testing problematic phenomena, and also in terms of how this process of testing plays out in larger patterns of work. That chapter focuses primarily on addressing my first research question, how software becomes a tool for exploratory work. In Chapter 5 I characterize the work of software production by examining a distinction that the Radio Group and their collaborators accomplish between *research code* and *collaboration code*. This focuses on understandings and enactments of software production work in the research lab, but also how exactly it comes to be articulated with the process of exploration. Chapter 4 and 5 together describe the basic aspects of a research software system, and Chapter 6 describes how this system develops over time, examining how research code piles up and 'graduates' to production code, as well as how the capacities of a research software system develop over time. Chapters 5 and 6 together answer my second research question, *how researchers maintain software as research problems and stakeholders change.*

Chapter 7 addresses my third research question, how researchers integrate software engineering practices into their research work, as somewhat distinct from the conceptualization of the research software system. This chapter is more about how change in practice around software was actually accomplished in a research group and its larger collaboration. I outline how these processes have sociality, rather than being dyadic, personal choices, and that scientific work is remade in various ways in the *integration* of software engineering practices. Chapter 8 is dedicated to connecting these findings back to the literature on scientific software, computer supported cooperative work, and science studies.

Chapter 2:
Background

This study marks out scientific software as a concern for the social study of science, for studies of collaboration and organization in the sciences, and for software engineering. Here I will draw on a number of different ways that these different disciplinary interests have approached (or could approach) scientific software as an object of concern. I first cover the fairly long-running discourse in software engineering scholarship around scientific software as a software engineering problem, and then move on to concepts of anomalies and unplannability in the science studies literature. I then cover literature on the topics of research infrastructures and problem formation, which form the backbone of my analysis of research software systems.

2.1 Scientific Software

Alongside and connected with movements for open science and reproducibility in the sciences, there has been a long-running effort to change the way researchers develop software. This effort in fact contains a variety of related, but distinct problems. These include robustness and reuse (Hong, 2014; Marshall et al., 2010), reproducibility and replication (Feinberg et al., 2020; Stodden, Krafczyk, and Bhaskar, 2018), sustainability (Carver et al., 2021; Ram et al., 2018), openness (Howison and Herbsleb, 2013), and productivity (Wilson, 2006) among others. Efforts to promote these characteristics of research software have also taken somewhat different approaches. One approach is to promote better software development practice amongst scientists. This involves the advocation of best practices (Wilson et al., 2014; Kelly et al., 2009), training researchers in software development (The Carpentries, 2021; Wilson, 2006a), and increasing recognition and citation of software work in the sciences (Hettrick et al., 2014; Katz and Hong, 2018; Du et al., 2021; Howison and Bullard, 2016; Katz et al., 2020). A second approach, which overlaps the first, is to create positions and career paths within academia and

national labs for people who develop research software (or do other kinds of computing work) as their primary activity (Berente et al., 2017; Baxter et al., 2012; Cohen et al., 2020; Carver et al., 2021; Sims, 2022). Here I will use the term “software engineer” for individuals who have a strong training in software engineering practices and whose primary output of their work is robust software. I will use the term “software developer” as a more general category describing anybody who does a significant amount of software development in their work. A software developer may also be engaged frequently in more exploratory kinds of work, although the term developer does designate a situation where effort is being made to build enduring software products.

Given how difficult these efforts have proved to be, there has been some reflection amongst software engineering scholars on what exactly is characteristic about science, and why it presents a different or challenging situation for software engineering methods. This has produced a large number of characterizations about scientific software development, such as that researchers cannot specify their requirements in advance, that they lack “oracles” against which to test their software, that they are often self-taught, and that they typically develop for their own case first and only later think about broader applications (see Heaton and Carver, 2015).

Two related themes which pervade these accounts of scientific software development are the oracle problem and the absence of up-front requirements. The oracle problem is a situation in which there are no outputs or gold standards against which the software can be tested. Although this problem has a longer history in software development, the absence of oracles is frequently cited as a characteristic which makes it difficult to bring software engineering testing practices into the sciences (Kanewala and Bieman, 2014). The fact that scientists cannot specify requirements at the outset of development work is similarly cited as a particular difficulty for bringing software engineering methods into the sciences, and is cited as contributing to the

ad hoc or contingent working style of scientific software development. Heaton and Carver (2015) summarize this idea in characterizing the association between scientific software development and the Agile methodology:

> *"When scientific projects are investigating new science, they are not able to determine all of the requirements in advance. Therefore, they cannot effectively use plan-driven approaches. Instead, the development teams need a methodology that allows them to experiment with different solutions as the requirements are discovered. This methodology would have to include many of the characteristics of the Agile development methodologies developed by software engineers [2]. Scientific software developers support the observation that they generally use an agile development approach because they do not know the requirements ahead of time [3]"* (pg. 14).

In these accounts the absence of requirements is a basic characteristic of scientific work and that characteristic has broad ramifications for their ability to plan and structure their software development work. The reason that scientists cannot specify their requirements, or the conditions in which they can or cannot is not well explored, but their work, science, is maintained as an explanation of the necessary contingency of their development process. The oracle problem sows similar chaos with scientists' ability to test their software systematically: because the output of the software is precisely what is at issue, it cannot be provided as a definition of correct behavior on the part of the software. Drawing on understandings from social and philosophical studies of science (to be discussed shortly), the instrument or experimental system is intended to produce emergent phenomena, or "unprecedented events" (Rheinberger, 1992a). If the outcome of the operation of the instrument could be defined ahead of time then the operation would not be functioning as a way of developing the researcher's understanding or creating new scientific objects.

Requirements engineering is in fact one place where social science approaches have made inroads into software engineering methodology, especially around the notion of requirements elicitation (e.g. Jirotka and Goguen, 1994). In a series of studies in the 1990s, a number of researchers brought ethnomethodological understandings of plans and the situated nature of

work into understanding requirements engineering. Goguen (1993) and Pinheiro and Goguen (1996) argued that requirements emerge over the course of a project, and that they are negotiated between people and in relation to local activities and resources. They and others also argued for the importance of naturalistic inquiry for understanding requirements because of the need to understand implicit knowledge (Jirotka and Luff, 2006). Despite its recognition of the emergence of requirements (rather than existing pre-formed at the outset of development work), much of the emphasis in this line of research was on this issue of eliciting tacit knowledge, rather than on how requirements develop or change over time. Under that concern, these studies have emphasized the importance of understanding requirements or needs within researchers' own perspectives and work contexts (Darch et al., 2009).

Some scholars have put forward more cohesive models of scientific software development. Segal's (2007) early work describes scientific software as "end user development", a situation or way of working with software that is part of a broader discussion in software engineering (Ko et al., 2011; Nardi, 1993). The idea of end user software development typically describes someone who is developing software as a means to a personal end, for the sake of accomplishing a particular goal, rather than creating robust software intended to be used by others. However, Segal (2007) pulls a number of other characteristics out of this kind of development, such as an iterative or evolutionary development process, the prioritization of consulting a network of colleagues over consulting documentation, the tendency to not take testing seriously, and the common situation of software which was initially designed for a single individual purpose suddenly needing to be made more robust for a wider set of use cases. These approaches seek to characterize scientific software development as a kind of software development, but do not attempt to provide broader accounts of scientific work into which those development activities must fit. They do involve, however, case studies or personal engagement with researchers and their day-to-day work.

Kelly's (2015) model, which she terms "risk averse software development", similarly looks for science as a model of software development, but it also eschews software development methodologies as they are traditionally understood for an understanding that takes scientists' practical goals more seriously. In particular she emphasizes that for scientists the outcome of development work is not working software but rather new knowledge. She emphasizes this in a series of case studies:

> "After the new module is running, the scientist may code certain pump parameters to be calculated and plotted in order to compare to plant data. Each activity increases knowledge in one or more knowledge domains. Nowhere is the software considered the end product. The software is a part of the integrated acquisition of knowledge from all knowledge domains. The end product is the scientist's increased knowledge moving him/her towards an answer to the scientific question" (Kelly, 2015).

Kelly also responds against characterizations of researchers as not testing or attending to quality in software. Her model in fact emphasizes scientists' aversion to risk, and describes some scientists' "relentless" pursuit of anomalies in their software (Kelly, 2015, pg. 54). Lastly, agreeing with Sletholt et al. (2011), Kelly argues that researchers do not operate with hard divisions between activities of requirements gathering, design, testing, and development, but rather do all of these things together in the process of their work. All together, this is a model of "software development as knowledge acquisition" (Kelly, 2015), which focuses on aversion to risk (in the consequences of the results produced by the software) and different "domains" that drive development work.

One benefit of Kelly's approach is that it attempts to situate software production in scientific work rather than situating scientific work in models of software production. Building on Kelly's argument, Paine and Lee (2014; 2017) emphasize the importance of the scientist being "in the loop", and examine the use of plots as a way of encountering and testing many parts of a layered software infrastructure. Goble et al.'s (2013) notion of "knowledge turns" targets the process of iteration in scientific work. Enabling the rapid sharing of software workflows would

enable researchers to more quickly build off of each others' work, thereby improving a critical dynamic in scientific work processes. Howison and Herbsleb (2010), too, examine scientists' own understandings of correctness in software (rather than taking formal definitions of software correctness) and found that they are deeply embedded in the social dynamics of the production of software: the fact that the software is widely used, the researcher's own familiarity with the code, or because it was validated against the results of other approaches. These studies pay close empirical attention to software components and their materialities, but their object of study encompasses scientific work. As Paine and Lee (2017) argue, the creation of scientific software is also the doing of scientific work.

Another line of research in CSCW and organization studies have looked at scientific software production in terms of incentives. These studies emphasize the amount of labor involved in producing software and the rewards that researchers reap from it. For example, Trainer et al. (2015) outline all of the various tasks researchers have to undertake in order to make their software into a "community resource", which range from bug fixing and managing dependencies to mentoring and writing tutorials. Howison and Herbsleb (2011) find a variety of models of software development across three cases, which vary particularly with regard to the incentives and rewards structures surrounding software development work. Across these models they identify tensions between the kind of maintenance and support work that software requires and the particular process by which researchers garner reputation through publishing:

> "The bifurcation between reputation for a software publication and citations derived from software use, shown in Figure 2, has potentially negative implications for direct collaboration on software between scientists. The authors of the initial software publication are frozen in time at its publication. Contributions to maintenance and support by others, while crucial for the software's usefulness and citations, are not rewarded by citations to the original paper. This means that such contributions are hard to turn into sources of academic credit for the later collaborators" (Howison and Herbsleb, pg. 521).

This attention to incentives comes up here in the context of researchers' contribution to software work, but it becomes salient also in discussions around making positions and careers for research software engineers in academia (Carver et al., 2021).

The idea of software as a "community resource" (Trainer et al., 2015) is another theme which pervades the literature on scientific software development. The observation that researchers often build software initially for their own purpose without considering other use cases is often noted in case studies (Heaton and Carver, 2015), and is associated with scientific software development efforts being *ad hoc*, rather than systematic across applications and use cases. Reusability is one of the primary problematics outlined in the problem of research software, and it is framed both as a problem of enabling the reuse of software among researchers in a lab or group (Feinberg et al., 2021), as well as the ideal of creating robust, community-supported scientific software tools that can serve many use cases (Koehler et al., 2020). As discussed above, the overhead of building more robust software, considered against the incentives of scientific work, is often seen as an obstacle to making software reusable. This involves both taking on more tasks (Trainer et al., 2015), but also organizational change in terms of how researchers collaborate with each other and the kinds of skill sets present in research labs and groups (Neang et al., 2023).

The trajectory of this literature shows an increasing engagement with the details and customs of day-to-day work in research labs, as well as more subtlety towards how software development work in those sites might be improved, but there is still a lot that is not well understood. While the work of Kelly (2015) and others (e.g. Paine and Lee, 2017) correctly point out the use of more situated understandings of scientific problems in lieu of rigidly codified requirements specifications, the issue of how researchers form these understandings through process of collaborative work, and how they guide software development work over time, still matters and is not well understood. Moreover, in some places this situatedness is given as a reason why

scientists do *not* employ engineering techniques, but such techniques are seeing some adoption in the sciences, or at least they are in the field site considered in this study. This includes techniques that look very much like planned software development, such as semantic versioning, deprecation, and requirements gathering. So how are these techniques being taken up alongside the more contingent processes so often ascribed to scientific work? Moreover, models of end user development and risk-averse software development seem to account for the ostensibly ad hoc and unsystematic development processes of scientists, but do not describe how these are reconciled with the development of robust infrastructural or community-supported software tools, which do exist in the sciences. More importantly, the boundary between, and differing conditions of production of, these different kinds of software has not been well explored.

2.2 Software and problems of materiality

I describe above how software has become a particularly problematic object in the sciences, but it is important to point out that software has in fact become problematic for almost everyone who interacts with it, from software engineers to practitioners of law. There are a variety of reasons for this. The term software collects together a variety of things, it invokes long-running assumptions about categories of the digital or electronic, and software always travels as a multiplicity of interconnected statements. These issues can make software a somewhat intimidating object of study. It is, seemingly, too many things to hold together. However, I would consider my own view of this as a tempering one. The complexities of software may be particularly troublesome but they are not a complete departure from the problems of complexity that harry scholarship on other technologies and societal phenomena such as scientific instruments, cyberinfrastructures, or bureaucracies. I will argue that we can approach software by adapting old conceptual tools, by approaching it in relation to practice and by following and

taking seriously the tools that people have already developed for making sense of it. Although it has changed radically and repeatedly, software is, after all, a quite old category of technology.

2.2.1 Software as digital system

Some of these difficulties are the difficulties which adhere more broadly to categories of the digital or electronic. As McKenzie (2006) points out, software is often understood as being intangible, operating invisibly and producing results seemingly from thin air. Digital systems are in this way treated by both their promoters and their critics as a kind of “technological sublime” (Kirschenbaum, 2008, pg. 34). This is a quality it seems to extend to things it composes, such as “the cloud.” In part this is a matter of imperceptibility, that one cannot see the operations and movements of software as it runs, only input and output. In contrast to the media it has replaced western business settings, such as paper, certain affordances of grasping and folding have ostensibly disappeared (Sellen and Harper, 2003). There is also, however, a mathematical claim on software, that software is definable as sets of relations and operations (McKenzie, 2006 , pg. 4). This, too, bolsters the notion of software as intangible, that software is best considered as a disembodied abstraction.

Many scholars have pushed past this abstraction view of software by examining its materialities, but this has complicated rather than simplified software as an object. Some have focused on the materiality of the screen and interface of computing, whereas others, such as studies of digital forensics, have focused on the materialities of magnetic storage, spinning disks, and transistors (e.g. Kirschenbaum, 2012). These views of hardware do have the benefit of rendering the acrobatics of digital things as accomplishments of vast and complex hardware systems. The digital thing can move around the world in the blink of an eye only because of the satellites and cables that make it possible. It is not a “context free grammer” (McKenzie, 2006, pg. 5), but rather it can move from place to place because of the standardization of computing technologies

from storage systems to data formats. As in Latour's (1983) description of laboratory conditions, it is the ability to "extend the rails" (pg. 155) of standardized computing systems to all parts of the world that allows the software to move so freely.

There is also, however, the argument that we should not try to convince ourselves of the materiality of the digital by attaching it somehow to the hardware or hardcopy materialities of other things. Digital systems have materialities in their own right in the way people interact with them. Dourish (2022) argues that while investigations of analog media or computing via other materials are valuable, they look for materiality in hardware, and thereby affirm the digital as essentially intangible. By contrast he draws on Schön (1979) in framing a materiality of digital systems that plays out in interaction with digital interfaces and objects. This is only part of a much longer recognition of the materiality of digital things (Leonardi, 2010; Sepkoski, 2017) and the deconstruction of a divide between the digital and the material (Pink et al., 2020).

2.2.2 Software as multifaceted artifact

This brings us to a second issue with software, which is that the term in fact collects together a large number of different things. McKenzie (2006) describes software as a "multidimensional and mutating thing" (pg. 2), and a "densely populated neighborhood of relations" (pg. 18). In some situations when one interacts with software they may be navigating many documents of written text, or finagling with the phrasing or indentation of a particular programming language. In this context software is code, not only translating between different symbolic languages, but also in its materiality as a thing written in a document according to particular grammars. In this materiality, some of the most important innovations of the field of software engineering are technologies for the manipulation, differencing, and versioning of text, and contemporary work continues in trying to construct better histories and comparisons of changes made (e.g. Wittenhagen et al., 2016; Pham and Kelleher, 2025). Redesigning the mode of interaction

between the programmer and the written code is still an active area of study, including in the design of notebook interfaces (Rule et al., 2018) as well as the introduction of AI agents (O'Brien, 2025).

One may also engage software as it runs, interacting with a produced interface and the restrictions that the rules of program place on its operations: it allows you to search but not by facets, it won't let you use more than 250 characters, the buttons give no indication of action when you press them, and so on. This is a mode of interaction that has come to occupy the lion's share of attention in contemporary design and usability research. One may also be running software on a command line, trying one incantation and examining the output, before trying another approach with different keywords or parameters. This mode entails both the textual interactions of programming but also the constraints produced by the running program and the interactivity of its responsiveness. The design of calls and visibility of code elements at this level is a design space above and beyond code architecture (Stylos and Myers, 2007).

One may also be working with an algorithm. While the recent overgrowth of studies of algorithms has not focused to a great degree on materiality of the kind implied in programming or software use, they do imply interactivities in rule construction, categorization, and visibility (Glaser, 2014; Burrell, 2016). Moreover, while algorithms are most commonly written out in code as part of a larger software package to be run, they also accomplish certain kinds of independence from the software. As a set of rules, an algorithm can be written in different programming languages or it can be written in plain language on a piece of paper. The sociality that emerges in the construction and enactment of algorithmic rule systems can be performed in non-digital, pedestrian ways, such as on a walk (Ziewitz, 2017).

The point of outlining these different engagements with software is to recognize the ambition of the term software in the number of different kinds of endeavors and interactions it is actually

seeking to explain. Sometimes we talk about programs, sometimes about code, sometimes about algorithms, and sometimes about software. My goal is not to dismiss this multiplicity as some kind of failure of clarity or specificity: collecting these things together in an object that can be considered as a whole is part of the work that the word “software” does. However, reflecting on the landscape can help position my interest in this study. The most salient concerns that have emerged around scientific software in the last two decades are not about the larger social implications of screen interfaces and GUIs that were of interest in the 90s, nor is it usually the concerns about opacity or categorization in rule-based algorithmic systems. The problematic of scientific software does touch on these things but its central concern has more alignment with those of software engineering: the emergent materialities of specific code bases in the way they pile up, the way they resist or facilitate alteration, the way they become navigable and comprehensible to their maintainers, the cleanliness of their design. These are the concerns of the work of software engineering, as opposed to, for instance, the programming or use of software.

It is both very difficult and very easy to approach software in this regime. It is difficult because it encompasses many different interactions with software. We can look at this in a case described by Spencer (2015). He describes a dynamic of “brittleness” as it developed in a software tool for fluid dynamics modeling:

> “...when it does break it is hard to figure out exactly why. Then, when you have figured out why it broke, it is hard to fix it. And when you do implement the fix, there is a good chance of causing further problems. Brittleness is a somewhat loose term that captures the experience of coders struggling with working in what has become a very difficult and frustrating medium” (pg. 472).

This term “brittleness” emerges across specific practices of development and use. It does not just describe difficulties in tracing the activities of the software retroactively, but also in navigating the lines of code and finding the correct place to make changes. Brittleness connects

the way code develops on top of itself, the way it breaks down, the way it is opaque to inspection, and the way it is resistant to even small changes. It is an evaluation of software as a kind of material for doing certain kinds of things, and it is an evaluation of a specific piece of software that has taken on this materiality through a specific history of design decisions and organizational changes (a sudden expansion in contributors in this case).

Spencer's account also provides an immediate path through this complexity, however. The term "brittleness" takes the software in its full definition, as something that is written and also run, as well as inspected and shared. Moreover, it characterizes software in longitudinal terms. It has characteristics that play out over long-running efforts to develop it and maintain it. These characteristics take shape in relation to particular practices, goals, and interests. This is more than a consideration of software as having a "human dimension" (Turk, 2013; Wastell, 1999); it captures the software in action. They reflect broad interactional tendencies in the way people work with the software. They contain evaluations of multiple aspects of a software tool that have emerged in past interaction or are likely to emerge in future interaction. Spencer's (2015) term for this aspect of software is "workability", which actually situates this materiality in relation to the literature on models in the sciences. It forefronts "models", and in his case software, as something that somebody works with and interacts with. This too is close to Schön's (1992) characterization of a conversation with the materials of design.

Materiality on this level remains a difficult analytical object. How can we hope to characterize the quality of such a large and differentiated system? My answer to this is similar to what it has been for many other complicated systems that have come under study, and that is to follow the actors (Parmiggiani, 2017; Ribes, 2014: Latour, 2005) in their own attempts to make sense of such systems. Cultures of software engineering and science are both awash with characterizations of software as "clean", "flexible", or "robust", along with phrases like "gradware" or "kleenex code" to describe broad interactional potentialities of particular software.

These represent not only aesthetic evaluations or efforts to leverage control over programming practice (Fedorova et al., 2025), but also sophisticated efforts to make sense of the complex materialities and potentialities of software tools that are built and maintained in different ways by different groups. Software might be a complex object but people have significant capabilities of description, as well as deep experience with software as a material that has developed over 70 or so years. Our project is not to draw on these characterizations unproblematically but rather to take developed, emic understandings of complex software systems as points of departure for understanding those software systems in relation to people's goals, practices, and epistemic struggles.

I focus on this understanding of materiality because it is the understanding that is of most relevance to the development over time of software systems as tools for scientific research. Many of the concerns that are central to software engineering, and to recent discussions of scientific software, are this type of understanding. They include qualities of sustainability or robustness (Hong, 2014; Marshall et al., 2010; Ram et al., 2018) that describe how a large complex software system can be worked with, extended, shared and picked up by new developers, and fixed when needed. My attention to the materiality of software could be on the implications of interfaces in social life, or it could be in algorithms as the enactment and definition of rulesets, but it is not. As with everything else, working through the multifaceted character of software is not a matter of constructing a universal account of it, but rather developing an account that is connected with one's goals and interests.

2.2.3 Software as interconnected ensemble

A last point is that software always travels as many things together. We refer to software as an uncountable substance, never as one or two softwares. When we consider a given "piece" or "package" of software, we must consider that it can only operate in the way that it is supposed

to by drawing on other software packages or dependencies, not to mention the operating systems and other processing technologies that underlie it. Moreover, all of these interdependent systems change frequently, with systems altering their functionality or simply going defunct in their support. The maintenance of relations (between pieces of software and also between software development teams) is therefore one of the central problematics of software engineering, and I would argue that it is a non-trivial problem for scholarship on software as well (see also Cohn 2019). This is not an entirely separate issue from the point made in the previous section, but it is worth considering the dependencies between software projects and their developers as a particular kind of relational difficulty.

Howison and Herbsleb (2010) have broached this issue by referring to scientific software as an ensemble artifact. By this they mean that when a researcher runs 'the software' for some particular purpose, they may be running code developed for particular analysis as well as shared or common code for their particular research group, code general to their larger collaboration or field, as well as code associated with operating systems, low-level mathematics, and other highly general tools. Code from all of these different kinds of software run in any given analysis or test that a researcher might pursue. This gets to the heart of the analytical problem because when we refer to software as an instrument we may want to consider some specific, named "piece" of software (such as the Python library astropy in astronomy), but the actual extent of the instrument that needs to be considered encompasses the much larger array of dependencies. In seizing hold of one specific software package we end up pulling out an incredible tangle of software systems.

The first thing that should be pointed out is that, in this particular regard, software does not differ as much as we might imagine from other kinds of technologies. A centrifuge, which is a central example in Rheinberger's (1992a) description of experimental systems, looks like a self-contained box that sits on the laboratory bench and operates as needed. It usually has metal

panels enclosing its internal parts, and in the manner of most modern personal computers (Woolgar, 1990) the ‘user’ is not meant to access or fiddle with its hardware components. In other words it appears as a definable, discrete appliance. Nevertheless the centrifuge relies on a great many other production processes and technologies being in place in order for it to work well. Improvements in electronic motors and controllers made greater speed and regularity of centrifugation possible, which in turn developed the tool (and eventually the “ultracentrifuge”) as an instrument for specific scientific problems (Rheinberger, 1992a, pg. 320). This is not a one way influence, as the development of centrifugation technologies was motivated from early on by the developing work of biologists and chemists (Pedersen, 1976). Similarly the development of much larger instruments establishes new networks of interactions between scientists and industries (Galison, 1992). While software may be particularly complicated in the way it relies on many interconnections between distributed development projects, it did not invent relationality *de novo*.

This is not such an encouraging point, because it asserts that what was bad news for understanding software is in fact bad news for understanding many other kinds of things. However, once again, investigations of those other things have provided strategies for proceeding. I once again draw on the strategy of following the actors. This is an approach that Cohn (2019) takes to the problem of mutability of software systems. As she points out, the quickly changing nature of software is not just a problem for the STS analyst but also for the developers of the software themselves. Recounting an instance in which a systems engineer makes a larger group of developers “feel” their interconnections by moving a single file and having them assess who was affected by that simple change. Cohn’s point in this case is about the “timeliness” of software, a temporality of remembering and forgetting. However, it also demonstrates the relationality of software as an object of software engineering work. Remembering and managing interrelationships between people and developers is an essential

part of building software. If the analyst finds difficulty in tracing these relations they are not alone.

Scholarship and practice on software engineering provide an assortment of techniques for cutting up, relating, segmenting, modularizing, and productionizing software into discernable, interconnected parts. These range from notions of packages or libraries to the technologies of package management, versioning, and, as I will argue later, the unit test. Perhaps most prominent among these is the notion of the module itself. Parnas (1972) original notion of modularity was directly aimed at managing the difficulty of interconnections between parts of a software system through a strategy of information hiding. In this model modularization is a process of turning a large software system into distinct parts through the definition of interfaces, which enable each part to hide as much of its functionality as possible from other modules, limiting what each module needs to know about others. This is to enable one person to work on part of a code base without needing to monitor or understand all other parts of the code.

The module has taken on a variety of other connotations in organizing ensembles of software. Sanchez and Mahoney (1996) define modules by focusing on the interfaces. These interfaces define certain inputs and outputs, which provides a skeletal structure, or “information structure” for a software system. In this understanding the module is in fact defined as a conformance to standardized interfaces. The module is a component *“whose interface characteristics are within the range of variations allowed by a modular product architecture”* (Sanchez and Mahoney, 1996, pg. 66). There are therefore many different kinds of things which can serve as the same module, and it is precisely through this design that a modular system enables loose coupling in an organization: it allows people to work autonomously and concurrently on different parts of a system. Baldwin and Clark (2000) similarly emphasize that once a modular component has been shaped by standardized interfaces its content can be changed out without affecting the rest of the system.

Drawing out the notion of the module in this way is valuable because it captures a number of the approaches characteristic to the management of software relations in software engineering. In all of these cases modularity is a kind of graceful, facilitated forgetting, which is accomplished through the implementation of standards. Also, all of these considerations consider social or organizational relations and not just connections in the operations of non-human artifacts. Modularity is almost always about what people need to remember, who they need to communicate with, and who they depend upon. Modularity is also recursive in a particular sense in that its rationales guide the design of low level functions or methods in software systems, as well as the design of larger classes, packages, and repositories of software. The module, then, is one of our most important points of departure for understanding how developers come to understand and manage the relationality of software systems.

An immediate route forward is to consider the module as memory and as delegation. The effort towards information hiding is in tension with Cohn's (2019) description of a technique for keeping software present. A key part of the purpose of a module is to offload thinking and cognitive load onto a contractualized entity. One only needs to remember and understand the input and output of a module and not its intricate internal operations. In this line of thinking it connects with long-held understandings of the routine as a way of offloading cognitive load, or the artifact as 'external memory'; it is a program or script (D'Addario, 2011; Nelson and Winter, 1982; March and Simon, 1958).

The goal here is not to discard activities of remembering or activities of forgetting in favor of the other, but rather to emphasize that in the design and use of software its intense and shifting relationality is engaged in large part through strategies of memory, and that these strategies are not unilateral or definitive. It's worth pointing out here that the definition of the module recaptured in Latour's (1987) concept of the black box:

> “The word black box is used by cyberneticians whenever a piece of machinery or a set of commands is too complex. In its place they draw a little box about which they need to know nothing but its input and output” (pg. 3).

The black box, too, is about keeping parts of a complex system out of mind, but that notion opens up our consideration of what that entails. Modules are often discussed as an accomplishment, but the accomplishment is one of design, and once established it performs the separation of concerns well. However, the black box is established and maintained by networks of actors. In this sense when one attempts to understand the characteristics and materialities of an artifact they are really making sense of a more or less concerted network (Camus and Vinck, 2019). This is a way of reapproaching the module as a strategic design, but one which is accomplished in an ongoing way by a large group of actors. Moreover, it is a thing which can be reproblematized, reinvestigated, and its contract or purpose can shift over time. This is the point of departure that I will take in this study, to take the module or black box as something which is accomplished in an ongoing way and to examine the sociotechnical dynamics by which that is done.

It is worth summarizing my approaches to these different problems surrounding software. Software presents itself as the digital in the sense that it cannot be pinned down because it is ephemeral and intangible. Prior work provides a number of routes for avoiding this difficulty, and my own is to avoid the distinction between digital and material and consider the digital system as something that has materiality in the interactions people pursue with it. It enables and constrains, as it is commonly put. Software presents many materialities and modes of interaction. I have framed one of these for its value in understanding particular dynamics around research software systems. My conception of software need not account for all possible interactions. Rather, I am interested in particular understanding of the materiality of software as it develops over time, in its workability. I am not interested here in the societal implications of the interface or of code more generally. Lastly, code is difficult to pin down as a discrete thing. It is

always a bundle of interconnected things. I think this is not so different from many other technologies, a plane or a centrifuge. “Actors” have made many of their own segmentations of what constitutes a piece or a package of software, and they use them frequently. My challenge is not to accomplish a perfect or real segmentation of interconnected code, but rather to understand the ramifications of the many delineations that are made, as well as their benefits.

2.3 Anomalies and unplannability

Discussions about the oracle problem and the absence of requirements in scientific software development bear remarkable resemblance to longer-running discussions of anomalies and uncertainty in the sociology of science. Drawing on this other scholarship can do a couple of things for us. Firstly it can complicate our understanding of how researchers relate to, and attempt to deal with, unknowns. Secondly, it can provide us with some theoretical tools to think about the connection between planning and uncertainty in scientific work.

Uncertainty has been a common leitmotif in writing on science since at least the 1930s, but it has been characterized and mobilized in a variety of ways. Fleck (1979 [1935]), for instance, works against the definitive and clear-cut understanding of experiment:

> “If a research experiment were well defined, it would be altogether unnecessary to perform it. For the experimental arrangements to be well defined, the outcome must be known in advance; otherwise the procedure cannot be limited and purposeful” (pg. 86).

Fleck’s point is firstly that if one knows the outcome ahead of time then the experiment loses any kind of revelatory character it was supposed to have. One does not learn something new from performing it. He goes further than this, however, in arguing that many experiments are not framed in clear-cut ways nor are they clear in their outcomes. It is possible to identify what he calls a “heroic ‘crucial experiment’” (pg. 10), but these follow on a long trajectory of less definitive trials through which researchers learn to formulate their questions. Fleck highlights this aspect of scientific work in response to a kind of historiography, or simply a post-hoc

rationalization, which leaves out the messier exploratory work in favor of a later stage where trials are highly specified and the experiment performs a demonstrative role, indicating “yes” or “no” in some pre-formed discourse on a topic.

In a quite different context, Polanyi (1945) develops the uncertainty of science as a problem germane to its governance. In particular he argues that science has the potential to change its own criteria for acceptability:

> “Yet the degree of independence granted to the scientist may appear to be greater than that allowed to other professional men. A businessman's duty is to make profits, a judge's to find the law, a general's to defeat the enemy; while in each case the choice of the specific means for fulfilling their task is left to the judgment of the person in charge, yet the standards of success are laid down for them from outside. For the scientist this may not hold quite to the same extent. It is part of his commission to revise and renew by pioneer achievements the very standards by which his work is to be judged” (pg. 141).

Polanyi’s strong gendering of these professions aside, surprise and uncertainty are his primary methods of rendering science as a distinct endeavor, as independent and pioneering. He was frequently in discourse with J.D. Bernal’s (1939) Marxist, state-planned model of science, and for him the unpredictability of science was a basis for its autonomy from external direction. Because science was precisely concerned with the discovery of new things, only experienced, practicing scientists could sit in judgement of novel scientific work.

Positivist and post-positivist accounts of science had an uncomfortable relationship with uncertainty. The systematization or rationalization of science was a common goal, but it required also delimiting consideration to those things which they believed could be successfully rationalized. Popper, for instance, agreed that the future direction of scientific work can not be laid out ahead of time (1957), but was thoroughly engaged in the project of logically defining a process of scientific justification. For him, and for many others in a post-positivist framing, the uncertainty of science was safely relegated to a context of discovery, distinct from a context of

justification. Discovery was just not tractable to logical analysis (Popper, 1959; cited in Simon, 1973). In this sense scientific work was unplannable in the long term, but its uncertainty was safely quarantined from the basic logical processes that made science systematic.

More recently, Star (1985) stridently affirmed the centrality of uncertainty in scientific work, and Fujimura (1987) considers it as a central issue in constructing "doability":

> *"Basic science requires facing uncertainty on a daily basis, as many writers have noted. Uncertainty decreases doability because it inhibits researchers' ability to plan ahead, which means that much of the work is carried out on an ad hoc basis… When constructing and solving problems, then, scientists attempt to create other conditions – including abundant resources, clear division of labor and packaged tasks – which reduce uncertainty or alleviate its consequences"* (p. 276).

In a way not dissimilar from studies of research software, she connects uncertainty with *ad hoc* ways of working. However, she positions this uncertainty alongside efforts and strategies, undertaken by researchers, to structure their work or their tools in ways that help them mitigate that uncertainty. Looking at scientific work, then, one might see moments of apparent chaos, where researchers drastically change approach because results surprise them, or expected outcomes continually fail to appear. Such readjustments and uncertainties clash with the tight timelines, promises, and organizational responsibilities that the researchers Fujimura follows are working within. One might also see researchers engaging in the stringent standardization of tools and procedures, precisely for the purpose of mitigating uncertainty. These two phenomena, then, are not alternative characterizations of scientific work, but rather aspects-in-tension that together help characterize the workplaces she is observing.

Where Fujimura plays out the phenomenon of uncertainty primarily as an organizational phenomenon, Rheinberger (1992a, 1992b) situates it in a material-discursive process he calls an experimental system, described below. In his account epistemic objects, the characteristically undetermined objects of researchers' work, and technical objects, the

characteristically determined resources and instruments used in that work, are mutually constitutive, with technical objects providing a material means to ask questions and give shape to unknowns. It is the relationship between the epistemic object and the instrument that creates the experimental system. In almost all of these accounts uncertainty is a central part of scientific work, but it is in tension with researchers' systematic efforts to control it, mitigate it, remove it, or structure it.

More recently uncertainty has come up as a theme in discussions of efficiency in science. Peterson and Panofsky (2021) draw on the same lineage I have described above in arguing that metrics for efficiency in science cannot capture progress in science (or a rate of progress) because the unpredictability of scientific work means that it is not clear what progress will be. Such metrics would rely on clear means-ends relationships and these are under revision at the cutting edge of science. This bears some relation to Polanyi's argument, and to others, but is situated within a contemporary discourse around the quantitative measurement of scientific quality, progress, and impact. The authors further point out that the argument against efficiency relies on a view of the messiness of scientific practice, whereas arguments for metrics of efficiency can draw on a popular, public view of science as a sanitized and rationalized process. In this way there is a visibility dynamic to uncertainty, where it is accounted for or left out of narratives of science. This relates closely to Fleck's (1979 [1935]) observations about the post-hoc rationalization of experimental work, which also expunge the more exploratory aspects of scientific work.

This long-running leitmotif of uncertainty appears again in contemporary discourses around scientific software. As described in the previous sections, software engineers approaching scientific work identify the absence of settled requirements as a key differentiator of that work from typical software engineering contexts (Morris and Segal, 2009). This is also emphasized by

Kelly et al. (2011), who position this as an inherent quality of scientific projects that must be accounted for:

> "Science is often concerned with pushing the boundaries of known knowledge, so it's no surprise that scientists push the limits of all the tools they use. That is, they find new ways to break the tools. It also means they create new questions to answer" (pg. 9)

Other scholarship on scientific software shows a similar appreciation of uncertainty as an immovable aspect of scientific work. It is for this reason, among others, that Agile methodology has been recommended as a viable approach for scientific work (Easterbrook and Johns, 2009). This is our point of departure, then, the acceptance of uncertainty as inherent in scientific work but a remaining lack of visibility into exploratory software processes themselves. Having asserted that scientists cannot adhere to the image of a waterfall or phased methodology, how then do they proceed? What are the quotidian processes by which researchers write and run software without rigid plans, and how do they make those processes work? This means looking more directly at processes of experimentation and exploration: how they are structured, organized, and adapted. There are, luckily, a variety of concepts we can draw on to do this. I look at these concepts in the next section, through the problematic of research infrastructure.

2.4 Research infrastructure and infrastructural change

What is missing in accounts of scientific software is an understanding of generativity in the writing and running of software. That is, how writing and running software produces new understandings and new phenomena. Here I will first cover some of the literature on scientific instruments and experimental systems which provides one basis for looking at software this way, and then discuss some of the organizational literature on the performativity (and therefore generativity) of routines. Lastly, I will cover some of the literature on research infrastructures, which has looked closely at tensions between emergence in scientific work and the continuities of infrastructural systems.

2.4.1 Generative tools

The importance of considering generativity in the design and use of software tools becomes more apparent when we consider them to be scientific instruments. Scientific software is often discussed as software, rather than in the terms and concerns of instrumentation. However, the comparison is an apt one, and it is made occasionally. For instance, Baxter et al. (2012) make this connection in describing the importance of attention and meticulousness in work with scientific software:

> "Computational work must reflect the committed attitude of experimentalists towards caring about precise, professional, repeatable, meticulous work – no-one with the same casual attitude to experimental instrumentation as many researchers have to code would be allowed anywhere near a lab. This is striking considering how often research results now depend on software" (pg. 1).

This characterization is made en route to arguing for the importance of research software engineers in the sciences, and it focuses on a particular aspect of instrumentation, which is the sense of rigor, meticulousness, and exactness that "the experimentalist" must have towards their instrumentation if they are to trust their results. This is one problematic that surrounds artifacts when they are considered as instruments, and it is a common epistemic virtue in scientific discourse.

There is, however, another problematic that surrounds instruments, developed mostly within philosophical and sociological accounts of science, which is their generativity. In Klein's (2001) detailed historical accounts of the adoption of Berzelian chemical notation, she describes how the notation becomes a "paper tool". As Klein puts it:

> "...paper tools, like laboratory instruments, are resources whose possibilities are not exhausted by scientists' attempts to achieve existing goals, but rather whose applications generate new goals" (Klein, 2001, pg. 295).

While working with chemical formulas does not entail manipulating the substances themselves, it provides a concrete representational space in which interactions between substances can be modeled and experimented with. In working through chemical formulas the researcher is able to arrive at new conclusions and understandings of how substances might interact. More than that, this new way of representing and working with substances prompted new directions for research, such as the ‘substitution’ approach. In this way interactions with tools shape and redirect inquiry in the short term as well as at the level of the research program.

Importantly, this agency-in-interaction of the material tool is not just a matter of its “correct” representation of substances, but of its graphic terseness and its maneuverability. This is an aspect of tools and instruments that Spencer (2015) highlights in the context of software. He refers to this as the “workability” of software as a material for science. This draws on philosophical work on models as being material artifacts in their own right. Spencer draws on philosophical work by Knuutila (2005; 2011) in particular that points to the affordances and constraints (citing Gibson, 1979) of models as concrete, material things that a researcher works with. The constraints of a model can be productive for the researcher, argues Knuuttila (2005), if cleverly devised. This workability, then, is one of the things that makes a good model. Spencer’s account of brittleness in a large codebase comes to bear on epistemic work in precisely this way, as a material for science that brings with it certain kinds of workability.

This is a view that aligns with studies of artifacts in Computer Supported Cooperative Work (CSCW) and related fields. Schmidt and Wagner (2004) for instance emphasize that the artifact is not a passive recipient of mental models, words, or speech. As they put it, artifacts are not simply “containers of preconceived ‘mental constructs’” (Schmidt and Wagner, 2004, pg. 271), but rather the material form of the artifact is essential for the work that it does. Schmidt and Wagner (2004) draw heavily on the work of Jack Goody to argue this even in the case of written artifacts. The spatio-graphical character of a list, for instance, provides the vertical and

horizontal basis for ordering, as well as essential discontinuities and empty spaces. The character of artifacts is not just reflected in their designed shape, but also in the way that they persist across time and accumulate wear and tear.

This might add, also, the idea that the coordinative aspects of artifact use contribute to its workability. Artifacts operate between people and in one way or another they must become visible, touchable, or encounterable to multiple people in an ongoing way. Schmidt and Wagner (2002) put it in terms of states and persistence:

> "Material artifacts are publicly accessible. Their state can be inspected by other members. Where they are located can be observed by and made sense of by members. What others are doing to an artifact can be noted and made sense of. As persistent graphical objects, architectural artifacts and configurations of such artifacts can be visually taken in simultaneously, at a glance" (pg. 271).

This is in fact connected to the notion of a field of work, which Schmidt and Simone write about in discussing coordination mechanisms (1996). The actions of one person change the state of a field of work and the coordinative artifacts within it, and that change in state is sensible to others. These observations may seem obvious, but it is important to recognize that coordinative artifacts enact a field of work or any kind of site of coordinative action: they make the ongoing interactions of people and things tangible as they happen, and provide the field of potential points of response. This is an important aspect of the workability of artifacts for epistemic purposes as for other kinds of purposes.

Looking at this aspect of instruments (and models) highlights a different criterion for scientific software: whether its materiality effectively facilitates the generation of new problems and phenomena in scientific investigation. This is one of the primary goals of Rheinberger's concept of the experimental system (1992a; 1992b; 1997), which I will draw on closely throughout this work. Working primarily with examples from a 20th century experimentalist era in biology, Rheinberger develops a philosophical account of the "practical process" (Rheinberger, 1992a,

pg. 307) of experimentation and knowledge production. Departing from traditional models of theory, experiment, and instrument, he focuses on a concrete account of an “experimental situation” and “experimental system” (Rheinberger, 1992a, pg. 307). The experimental system has a couple of components, including most centrally the “scientific object” or “epistemic thing.” The epistemic thing is the object of investigation of the experimental system, and it is characteristically vague and poorly understood. If it were not, then the experiment would have no purpose. It is precisely the character of the epistemic thing to be a “question generating machine” (Rheinberger, 1992a, pg. 310). However it is not just that the epistemic thing is hidden or obscured, it is in the process of being materially defined by the experimental system.

The second aspect of the experimental system is what Rheinberger (1992a) calls the technical conditions, or the technological objects. A technological object, in contrast with an epistemic thing, is characteristically determined. They operate according to known regularities, and under the right conditions they function as question answering machines. That is, in well understood situations they answer well-formed questions. The experimental system, however, turns on the interaction between these two things, the epistemic thing and the technological objects. The technological objects materially define the epistemic thing in a new way, and in doing that they are engaged in a nontrivial interaction which is outside of their typical “boundary conditions.” As Rheinberger describes it, the technological objects contain the epistemic thing, both in the sense of embedding it and in the sense of restricting it. This is a somewhat ineffable interaction when speaking generally, but the core point is that the pursued objects of science are made, as in progressively materially defined, prior to the experimentalists understanding of what it is that they are producing. In certain contexts of use the technological objects act as intended, according to known rules, but in the experimental situation it operates in uncertainty.

It is important to note, also, that the technological objects and the epistemic thing are parts of a system, and they are functional designations rather than inherent material ones. They can also

switch places, so to speak, if once well understood parts of the system come under uncertainty. The epistemic thing, too, can come to be defined to the point that it feeds back into further (different) experimental work as a technological object.

A key aspect of this theorization is how these different aspects of the system interact over time through a process of differential reproduction. The term differential reproduction is intended to capture the fact that the experimental system intentionally produces difference, by which Rheinberger means that the whole arrangement remains open to the emergence of "unprecedented events" (Rheinberger, 1992a, pg. 323). However it does so within technical conditions (the technological objects) in a way that makes this difference coherent and recognizable within a process of reproduction. If the system simply produces unintelligible newness, unconnected with framing technologies that give it coherence, it does not serve its purpose as an experimental system. This is perhaps the most critical point for my analysis, that in this nontrivial interaction between epistemic thing and technological conditions, the experimental system becomes a system that can "give answers to questions which we are not yet able clearly to ask" (Rheinberger, 1992a, pg. 309). This is a somewhat mysterious statement, but it builds on Fleck's (1987 [1935]) idea that a great deal of experimentation is focused on producing unanticipated outcomes in response to poorly formed questions. Experiments that are able to shape clear "yes" or "no" questions occur at the end of the epistemic work, when the concepts in play have already been clearly defined.

This focus on the epistemic thing coming into being is an account of science focused on material interactions and low-level practices of phenomena development. For this reason it has found currency with studies of design. Ewenstein and Whyte (2009) in particular have picked up the concept in order to understand how visualizations leverage regularized techniques of representation to shape new interactions with as of yet poorly defined design ideas. They focus in particular on Rheinberger's (1997) notion that the experimental system presents spaces of

representation, in which poorly defined epistemic things progressively take on shape. Moreover, as in Rheinberger's account, they argue that these epistemic things, or epistemic objects in their phrasing, are provocative in the sense that they raise new questions and present possible directions forward. In their case of architectural design drawings they point to the example of a diagram where a floor plan is placed under a working representation of the floor above it. The underlying floorplan functions as a technological object; it is set and provides a stable reference point against which the floor above (the epistemic object) begins to take shape.

While Rheinberger's (1992a) and Ewenstein and Whyte (2009) are divergent in some ways, they both provide a couple of key conceptual moves that are useful to me here. Firstly, they focus on the formation of poorly defined things. These are concepts useful for talking about development and design work prior to it being well understood or specified. It is therefore a fitting tool for studying problem formation. They also both have a focus on material practice, by which novel things do not just emerge as thoughts or ideas, but rather they are actively made. The experimental system also brings us back to a notion of workability, although with some new parts added on. The experimental system defines the researcher's ability to maneuver, the kinds of questions they can ask and the kinds of answers they can produce (Rheinberger, 1992a).

2.4.2 Flexibility and change in work and organization

While I draw closely on these ideas of the generativity or performativity of tools and artifacts, I also need to account for flexibility or adaptation in longer running patterns of action, such as in routines and protocols. In doing this we can draw on a variety of performative accounts of planning and routine, as well as on specific studies of research infrastructure.

A central conceptual move in performative accounts is to position plans as resources for action rather than definitive descriptions of action to be taken (Suchman, 1987). In a similar vein Lynch (2002) describes the “procedural flexibility” of the PCR protocol in the laboratory environment, by which researchers use the representation of the protocol in the performance of the polymerase chain reaction (PCR) procedure. Of particular emphasis in their work is the idea that although the PCR procedure has been integrated as a “humdrum feature” of laboratory work it nevertheless requires implementation and discretion of the operator each time it is performed.

There are a number of different ways of considering how plans, routines, or artifacts become resources for the performative action. Lynch (2002) provides the analogy that a protocol is like a recipe, which is implemented at the discretion of the researcher. Ankeny and Leonelli (2016; Leonelli and Ankeny, 2015) provide the notion of the repertoire, which is the set of resources (data, tools, funding arrangements, personnel) that researchers can leverage in conducting new investigations. The repertoire also includes the know-how of putting these things together in productive ways. In using the term repertoire they draw in part on Faulkner and Becker’s (2019 [2009]) description of the musical repertoire as resources for the construction of a performance. Faulkner and Becker’s interest is in the way that musicians can “remember” large arrays of songs by remembering common harmonies and chord progressions and then assembling these resources into a performance. The actual performance is a “repertoire-in-action” (Faulkner and Becker, 2019 [2009], pg. 19) as they put it.

This analogy to music is used also by Weick (1998), who draws out a number of important aspects of improvisation from the example of jazz performance. Drawing on Berliner’s (1994) ethnomusicological work, he points out that the notion of improvisation does not imply complete chaos or novelty. Musicians are not pulling improvisations from thin air, but rather they are altering and departing from standard tunes in particular ways. This is a large part of the work that later accounts of repertoires do: they account for the fact that the improviser is able to

improvise precisely because of, and not despite, their training and learned patterns of performance. This is an essential aspect of the protocol, tune, harmony, or routine which we will return to below. Weick (1998) also points out that improvisation involves retrospection, the consideration of past outcomes in the projection of new actions. The improviser is not working from plans and looking ahead, but considering what they have just played, shaping new notes in relation to immediately past outcomes (see also Gioia, 1988).

Work on routines has also drawn on a distinction between performances and representations of action. Feldman and Pentland (2003) describe ostensive and performative aspects of routines, where the ostensive aspect is the represented structure or protocol of the routine. Like Cohen's (2007) "dead routines" these are representations or plans but they are not the routine as it is enacted by people. The notion of the dead routine, the representation of a routine, does not discard those things from consideration, as representations of patterned action are both important objects for understanding those routines as well as resources to be used in the routine itself (D'Adderio, 2003; 2008b). D'Adderio (2011) points out, for instance, that anthropologists and archaeologists reconstruct the actions of cultures and communities from their artifactual vestiges. However, new action, even when it seeks to conform to some artifactual constraint or representation, constitutes a new performed action, which opens the possibility for variation, adaptation, reimplementation, and so on. As Feldman and Pentland (2003) argue, even routine action can entail reflection and reinterpretation on the part of the person carrying it out. In this sense routine action can become a source of change in patterns of action.

It is important to point out that there are a variety of artifacts involved in routines (not just representations of the routine itself) and these too can create variation and change. D'Adderio (2011) points out that initial efforts to render routines as more than rigid repetition have focused on individual human agency to the exclusion of the agency produced by artifacts. Highlighting

people's agency helps account for the way that they perform established routines again each time, and in so doing they bring about a possibility of novelty and change. In this framing artifacts can take on a constraining role, where they primarily help enforce routine action as against the agencies of humans. In contrast with this understanding D'Addario emphasizes the need to look at the way artifacts play into the reproduction and change of routines.

Extending the use of Callon's (1998) terms, D'Adderio (2008a) describes a process of overflowing and re-framing by which action with artifacts reconfigures routines in an iterative way. Of significance in this characterization is the idea that action is always somewhere between total prescription by ostensive routines (representations of routines) and complete arbitration on the part of the human actors involved. In this understanding all action is performative but there are distinctions in the influence that plans or protocols have on an actual performance. The performance of an action can, for instance, make a process more like the rule that describes it (MacKenzie, 2003), or it can make the process less like the rule that describes it (D'Adderio, 2008a).

A similar discussion has occurred around the notion of the boundary object, introduced by Star and Griesemer (1989). A boundary object is an object that inhabits several different social worlds, but which satisfies the "informational requirements" (pg. 393) of each of them. In this capacity the boundary object is malleable enough for people to adapt it to local needs and conditions, but it is also robust enough to maintain identity across sites of use. This is a conceptual intervention in the framing of translation as a particular, central actor's entrepreneurial efforts (e.g. Law, 1987), focusing on coherence as something produced by n-way sets of translations undertaken by different actors. Lee's (2007) concept of the boundary negotiating artifact extends the notion of the boundary object by focusing on how artifacts not only embody or facilitate stable interactions across social worlds, but serve as means for destabilizing protocols and pushing or negotiating practices between actors and social worlds.

The boundary negotiating artifact makes a broader conceptual point, but a large part of its value is in the characterization of specific kinds of boundary negotiating artifacts. Looking at specific artifacts and how they are mobilized provides a way of examining the destabilization of routines or the pushing and asserting of new practices or coordinative arrangements in detail. In Lee's (2007) discussion, these include, for instance, inclusion artifacts, by which new ideas are proposed and evaluated to a group, and structuring artifacts, by which people would assert ordering principles for activities or give shape to others' work. As with other accounts discussed above, the materiality of artifacts has agency in negotiations and practices undertaken by groups of people, but their role is not deterministic. It is situated in the way they are leveraged by people with respect to others with whom they are coordinating.

These accounts of performativity verge on another issue, which has been frequently discussed in studies of research infrastructure, and that is the issue of flexibility. The notion of flexibility in a given relationship also bears on the relationship between artifact or protocol and the actual performance of a thing. Like D'Adderio, Schmidt and Wagner (2002) also account for the complexity of the relationship between artifacts and flexibility in action. In some cases flexibility is accomplished through under specification. Architects, for instance, can adhere to particular standardized line widths used across companies and sites of design, or they can alter these standards to better address the needs of designing a particular building (Schmidt and Wagner, 2002). In this way flexibility is a matter of the ability to tailor particular artifacts (architectural drawings and drawing techniques) to the needs of one case over another. They highlight sketches as being open in exactly this way, making themselves amenable to commentary and new ideas. Rigidity can be associated with the imposition (appropriate or stifling) of a standard in a site of work, whereas flexibility can be a source of problematic ambiguity or a necessity for making an artifact fit with a local practice.

One important point, however, is that these two characteristics are not necessarily at odds, nor are they exclusive. In relation to a specific practice a coordinative artifact can be both rigid where it needs to be and flexible where it needs to be. This is pointed out also in Bietz, Ferro, and Lee (2012), who argue that standardization is in fact an important part of creating flexibility because an artifact (a database) which is standard in the right way frees up time and enables the work of a specific job (pg. 909). Moreover, flexibility is not accomplished *despite* the durable, ordering character of artifacts, but rather where an artifact can be leveraged for multiple practices, it is useful for each one precisely *because* of its durable ordering character. It is the relationship with practice that is flexible across cases. This point concerns standards rather than artifacts, but it nevertheless aligns well with D'Adderio's (2011) effort to avoid taking artifacts as inherently constraining against human agency. It posits flexibility as a more complex kind of accomplishment between the rigidities of material things and the goals and intentionalities of human actions.

Discussions of flexibility have been central to discussions of research infrastructures, particularly how they grow and change over time. As with the instances of flexibility given above it is important to be clear about what exactly is becoming flexible. Ribes and Polk (2014) outline three different kinds of flexibility that concern change in different kinds of things: technoscientific (research objects, methods, and instruments), sociotechnical (organization, coordination, and collaboration), and institutional (funding and regulatory regimes). Ribes (2014) puts forward the idea of the kernel of research infrastructures as the core set of services that they make available to support ongoing research work as well as the work of maintaining those services over time. Key to this notion is the idea that maintaining such services is a matter of ongoing renewal or regeneration of infrastructural services in the face of changing scientific objects, or technoscientific change. Critically, the work of making core services more robust do not necessarily inhibit changes to new objects of investigation, but rather, through repurposing,

those developed systems can become resources for, and help constitute, new objects and programs of research (Ribes, 2015). In this sense what is robust about an infrastructure is not a set, unchanging foundation of technical services, but rather a kernel which is maintained precisely through its extension and re-adaptation.

Lee et al. (2010) similarly look at stability through adaptation in cyberinfrastructures, but they look in particular at stakeholders and the management of relations in human infrastructure over time. This is not just a management of relations between people invested in a project, but a kind of “relational maintenance” (Bietz et al., 2012) that sustains the working order of a cyberinfrastructure. Thus shifting stakeholders and developments in metagenomic science prompt changes to the service offered by the cyberinfrastructure, which in turn prompt new changes to the human infrastructure (Lee et al., 2006) that supports and sustains that system. Part of this relational maintenance is the active construction of a “community” in terms of understood stakeholders and relations that the cyberinfrastructure maintains with users and other invested parties (Ribes and Finholt, 2008; Lee et al., 2010).

There are a number of takeaways from this literature on research infrastructures. The first is that flexibility and rigidity are locally-produced dynamics which can mean very different kinds of things, from flexibility in research programs and objects to flexibility in the relations maintained with stakeholders. Second, the relationship between flexibility and rigidity is not a simple tug-of-war between “rigid” things–artifacts, plans, and standards–and the autonomy of the individual. Standards or artifacts (or standardized artifacts) can certainly impinge on the flexibility of certain kinds of work, but they can also be the very basis for producing flexibility in work.

2.5 Problems and problem formation

In this study I approach scientific software development as problem formation. Examining problem formation is useful because it provides a processual, pragmatic account of what

research problems are, and how they relate to the development of the tools built to address them.

Problem formation, or problem selection, has been a central interest of the sociology of science at least since Merton's (1938) writing on science in 17th century England. Merton tracked change in scientific interest using counts of publications in different fields (e.g. "physical sciences" and "biologic sciences") in the *Philosophical Transactions of the Royal Society of London* (pg. 403). Although Merton argues that problems are affected by "internal" developments in a field, he is particularly interested in the influence of external "cultural values", such as shifts in religious belief. Problem selection or problem formulation were again important topics in the 60s and 70s in debates about the growth of knowledge (e.g. Kuhn, 1962; Lakatos, 1970; Laudan, 1977). In general these early discussions were interested in research traditions, paradigms, and programs; they were on an order much larger than what I intend to tackle here. They were also, for the most part, interested in problems as parts of a history of ideas, without much interest in the tools or day-to-day work of research.

For the purposes of this study, Fujimura's (1987) work on "doable" problems is an important point of departure for a couple of reasons. Firstly, for Fujimura a scientific problem is a thing that has organizational and material dimensions, and this helps us understand the connection between anomalies or uncertainty, described above, and the organization of scientific work. She describes the development of an oncogene research problem at a commercial company, and focuses on the way that alignment must be achieved between specific experiments, the delegation of work amongst technicians in the lab, and the larger goals of the "sponsors" in the organization. In such an example the problem that ends up being engaged is the one which can play out in experimental interactions with genes (and with at least "good enough" commitments to scientific integrity), that can be effectively delegated amongst members of a lab or other people who can be brought in, and that conforms with the company's general set of goals and

holds promise for those goals. Along with earlier laboratory studies (Latour and Woolgar, 1979), this moves the analysis of science away from a world of interacting ideas and theories towards the day-to-day material and coordinative interactions that constitute scientific work.

The second reason that Fujimura is a useful point of departure for this study, is that in moving from problem selection to problem formation or construction, Fujimura (1987) begins to frame problems as things which take shape, or are shaped by the actors involved, rather than existing pre-formed to be chosen amongst. This framing has roots in pragmatic and interactionist understandings of knowledge and inquiry. Dewey (1939), for instance, makes the definition of a problem an essential part of the process of inquiry:

> "A problem represents the partial transformation by inquiry of a problematic situation into a determinate situation. It is a familiar and significant saying that a problem well put is half-solved. To find out what the problem and problems are which a problematic situation presents to be inquired into, is to be well along in inquiry" (pg. 90).

Here inquiry does not begin with the search for solutions to a pre-formed problem, but rather much earlier, with the putting together of a coherent situation and one's one stance or interests towards it. To assume that an indeterminate situation is already problematic, before one has done any work to formulate it as a problem, would be proleptic: it posits the existence of something before it was there. Moreover, the shape of the problem affects the shape of solutions presented or actions taken:

> "To mistake the problem involved is to cause subsequent inquiry to be irrelevant or to go astray. Without a problem, there is blind groping in the dark. The way in which the problem is conceived decides what specific suggestions are entertained and and which are dismissed; what data are selected and which rejected; it is the criterion for relevancy and irrelevancy of hypotheses and conceptual structures" (Dewey, 2023 [1938], pg. 90).

With this understanding, if we consider scientific activity in the context of justification, where hypotheses are being tested or demonstrated, we have already missed a great deal of the thinking and investigation which has given shape to the entities involved in a situation, the larger

reasons or goals of the investigation, the specific hypotheses, and what constitutes a good or good enough test.

This approach moves our discussion away from the common preoccupation with a context of justification (sites where scientific ideas are demonstrated, proved, or tested) towards the genesis of scientific problems and ideas, which were traditionally dismissed as part of a context of discovery. As Leonardi (2012) puts it, it treats problems as constructions in their own right. Even in contemporary discussions where the model of contexts of discovery and justification hold less sway, there is a bias in broader discussions about science towards issues of peer review, retraction, replication, and other activities of checking or proving. While I do not want to give the impression that those lines of investigation are not valuable, the examination of problem formation should be recognized as a much-needed, and complementary, line of inquiry.

Part of what is pragmatic about the above stances is that the criteria for evaluating some given statement is not in the statement itself but in the situation and the framing of a problem; it is not that people produce statements or understandings within situations that might be correct or not correct, rather “correctness” is produced in relation to a situation. In just this way Clarke and Fujimura (1992) are concerned with the “rightness” of tools, which they render in relation to the shifting career trajectories, timelines, resource pools, and materials of scientific workplaces. This brings us to the idea that whether tools are useful, whether their material design is appropriate, is not something that can be evaluated against a standing set of criteria for tools in general, but rather must be evaluated with reference to the situations and problems as they are constructed by people, in my case by researchers in astrophysical labs and collaborations. The “rightness” of the tools is produced within those situations of work, and it is produced by the actors involved. This is why I would argue that examining problem formation is a good way to get at the issue of how software engineering methods come to be useful for science, because it

is there that the usefulness of software tools is being worked out and rationalized in connection with developing research problems.

These kinds of pragmatic or process-oriented approaches have been used to examine technology design and use in a variety of contexts. Reminiscent of Fujimura's (1987) notion of "do-ability", Dew et al. (2019) describe how people articulate "printability" in 3D printing. They describe how people think about printing projects in relation to a variety of other concerns, such as larger work processes and timelines. Moreover, experienced practitioners developed an embodied sense of printability: they developed a sense of the probable success or failure of a given print given the different concerns that would come into play, such as the compatibility of different filaments, support structures, temperature, and the effects of particular machine malfunctions. On an organizational scale, Leonardi (2012) describes the negotiation of different problem constructions at an auto-manufacturing company, and how different stakeholder groups had to come to see a given design as a technology that could solve their respective problems. This required formulating and formulating the problem, eventually casting it as a solution to a "standardization problem" that was in fact understood quite differently by the different stakeholders involved. As he writes,

> "These findings suggest that new technologies aren't born out of a simple stepwise progression in which developers identify a set of problems and subsequently develop a new technology to solve them. Instead, multiple groups in an organization actively identify problems that appear important and manageable to tackle while simultaneously identifying technologies that validate the existence of the problems by rendering them solvable" (Leonardi, 2012, pg. 88)

Leonardi also emphasizes the sense of inevitability that can take hold in organizations around the proper design of a technology, and the implied problem that it is supposed to solve. Part of the benefit (for Leonardi and for others) of looking at problem formation or construction is that it cuts a third route apart from strong social constructionism or technological determinism. Neither the problem nor the proper technology design is taken as necessary or inevitable.

Studies of scientific problems do not only consider how the researcher or community shape problems, but also how problems shape communities. Rheinberger (1992a; 1992b) and Fujimura (1987) are both interested in how particularly generative problems 'take off' and become default approaches in a field and are leveraged as a template for conducting work elsewhere. In a similar approach Kohler (1994) looks at Drosophila (a variety of fruit fly) as a "breeder reactor", a model biological system which rapidly developed new mutations, and therefore new research questions and directions. As a productive system for generating research questions, and ones that could be materially investigated with an organism that survives well in academic environments and timeframes, Drosophila traveled from their lab of origin all over the field, such that almost every biology department had to have some work on Drosophila going on.

Taken together these observations give us a couple of basic theoretical guidelines. In examining the process of software design in the sciences, we should not look for underlying problems, provided by nature, which pre-exist people's activities of development, nor should we be looking to particular technology designs to determine research questions or programs in a strong sense. Instead we can look at processes by which technologies and questions are formulated in relation to one another in a process of problem formation. Moreover, this literature prompts some interesting areas for further examination, such as how problems are formulated between researchers, and how tools become useful across problems.

Chapter 3:
Methods and methodology

This study looks at the development and use of research software in the context of a research lab in the field of reionization cosmology. The study takes the form of a laboratory study, using ethnographic techniques to examine the practical activities and contexts of knowledge production. My engagement with the group began in 2019, and has continued off and on for 6 years. Analysis involved a qualitative coding process based in grounded theory and drew on interview data as well as observations and documentary evidence. I will first discuss some of my methodological commitments (section 5.1), and then turn to the field site (5.2), data collection, and analysis (5.3).

3.1 Methodology

My general methodological approach is theory-generative, drawing on grounded theory (Charmaz, 2014) and abductive analysis (Tavory and Timmermans, 2014), but draws also on case study design and ethnographic techniques in order to structure and engage my field sites. I will outline my general approach first, and then discuss a number of methodological issues. Grounded theory is a widely popular methodological approach rooted in pragmatic and interactionist sociology (Clarke and Star, 2008; Bryant and Charmaz, 2007). The goal of grounded theory is theory generation, rather than theory testing, and it provides a set of methods that are intentionally designed to help the researcher in this generative process by forcing them to engage the data repeatedly at a low level, make comparisons across instances, and reflect on assumptions and labels as they are being produced.

I draw on many of the classical techniques of Charmazian grounded theory methodology (Charmaz, 2014), including initial and focused coding, constant comparison, memo writing, and

theoretical sampling. However, I also draw some methodological insight from abductive analysis (Tavory and Timmermans, 2014), which is a conceptual offshoot of grounded theory. In particular this involves engaging literature more in the process of analysis and using techniques like alternative casing in order to look for disjunctures between these theoretical frameworks and the data. Abductive analysis argues that *"An abductive inference involves making a preliminary guess based on the interplay between existing theories and data when anomalies or unexpected findings occur"* (Timmermans and Tavory, 2012, pg. 179). In this account the researcher must be conversant with many theories, but consider them in close connection with the data. Repeatedly revisiting the data in the coding process is a strategy to *"...increase the resistance of the phenomenon to our interpretations"* (Timmermans and Tavory, 2012, pg. 175).

As a last point, Grounded theory takes a stance, held by earlier symbolic interactionist thinkers, that it is essential to understand people's objects as they themselves see them, because it is on these understandings that people take action (Blumer, 1986). Supplanting peoples' conceptions of things with the researchers' or with some preconceptions existing in the scholarly literature, would be a failure mode for a study of this kind.

This study's primary impact is intended to be a matter of sensitization. It is a reasonable question how we are intended to draw broader meaning about scientific software from a model of it developed in one context, within a particular research lab. It would also be dishonest of me to claim that the ideas that I develop here do not seek to 'generalize' in some sense of that term. I have developed the notion of the research software system because I believe that it can be helpful to a broader set of activities around scientific software, and in fact in other contexts of exploratory work as well. The goal of this study is to develop sensitizing concepts, which can guide one in where to look, but does not tell them what to see (Blumer 1986). Charmaz (2006) describes the use of sensitizing concepts in this way:

> “Guiding interests, sensitizing concepts, and disciplinary perspectives often provide us with such points of departure for developing, rather than limiting, our ideas. Then we develop specific concepts by studying the data and examining our ideas through successive levels of analysis [...] In short, sensitizing concepts and disciplinary perspectives provide a place to start, not to end” (pg. 17)

For Blumer this was in opposition to so-called definitive concepts, which are understood as leading directly to some empirical content of an instance, what Blumer called “explicit objective traits” (Blumer, 1986, pg. 150). This would include something like a benchmark, which is expected to measure something that is stable in its empirical form across sites of investigation. For Blumer, using concepts in this way imposes an abstract theory onto empirical cases, rather than taking empirical cases on their own terms.

It is within an interactionist understanding of social life, with its strong ontological contingency, that it makes sense for the concepts we pursue to be sensitizing and not definitive. For Blumer, as well as for George Herbert Mead, interaction is not just a complicated outcome of some more fundamental structural or psychological features; interaction is an ontologically productive force. It is *“a formative process in its own right”* (Blumer, 1986, pg. 66), through which reality is recast. This means that the concepts that we grasp for in research will never reach a point that they exhaust all future concerns that we might have with new emerging cases of social life. What the scholar is after in each new case is what is distinctive about it. Blumer provides the analogy of handling a strange object:

> “The prototype of inspection is represented by our handling of a strange physical object; we may pick it up, look at it closely, turn it over as we view it, look at it from this or that angle, raise questions as to what it might be, go back and handle it again in the light of our questions, try it out, and test it in one way or another [...] Such inspection is not preset, routinized, or prescribed; it only becomes such when we already know what it is and thus can resort to a specific test…” (Blumer, 1986 pg. 44).

What is significant about this object under inspection is its strangeness. As in Dewey’s (2023 [1939]) account, if we already knew how to approach it, we would have already defined the

problem as well as its definite shape; It would already be familiar to us. Asserting a situation of this or that kind is a product of inquiry, not a starting point. Reaching for this or that procedure for making sense of a situation asserts what the character of the situation is: in some sense it tells us what to see. By contrast, the sensitizing concept might guide us in our inspection, but because new cases are fundamentally distinctive, we must understand the relation as one of sensitivity rather than definitive understanding. The bad news brought by this underlying view of the world is that we, the researcher-organisms in Mead's terms, will never reach concepts that are once and for all definitive of the phenomena that we will care about. The good news is that it means sociologists will always have fresh business.

The sensitizing concept should also be reformulated in the face of new empirical data, and Blumer argues that it can be refined and tested. This is not a matter of confirmation or disconfirmation at a single point of trial, but rather establishing a close, ongoing relation with the empirical world. In this process engagement with empirical data is not a process of 'checking', it is a frequently repeated process by which the concept is intentionally expanded and reformulated. Although the sensitizing concept does not provide a prescriptive definition, it can be formulated and reformulated, communicated, and evaluated against new cases. It is not formulated through the creation of benchmarks and measures, but rather through *"exposition which yields a meaningful picture, abetted by apt illustrations which enable one to grasp the reference in terms of one's own experience"* (Blumer, 1986, pg. 150). I describe some potential areas where these sensitizing concepts might inform future investigations in Chapter 8.

In Chapter 8 and elsewhere I discuss sensitizing concepts as an *outcome* of my investigation, whereas they are usually discussed as a starting point for an investigation. My purpose in doing this is not to play gatekeeper as to the later usage or readaptation of the concepts I develop here. It is a characteristic of this methodological stance that I cannot exhaustively anticipate or control how these concepts might become productive in future research. Nevertheless I can

provide some potential directions and suggestions. This benefits the reader who is considering how they might use these ideas, but it also helps define and clarify the concepts themselves. For instance, pointing out that the research software system focuses our attention on articulation work helps clarify what the research software system is, a system of heterogeneous, articulated processes. Moreover, it helps convey my own understanding of the potential breadth of these ideas, such as the fact that they are not inherently bound to a category of "scientific" activity.

One issue that has been raised around the use of grounded theory methodology and interpretive coding methods more generally is the transparency and systematicity of the coding procedures and analysis process (Stol et al., 2016). Grounded theory emerges from a strain of sociology which has pragmatic commitments to the idea that methods cannot be definitive of the phenomenon under study. In a symbolic interactionist perspective, interaction in the world is constitutive (in a strong sense) of new things (Blumer, 1986), and so the researcher cannot enter an investigation of new things with a universal or fundamental method that would become definitive of the thing studied. Blumer makes this point clearly:

> "The prototype of inspection is represented by our handling of a strange physical object; we may pick it up, look at it closely, turn it over as we view it, look at it from this or that angle, raise questions as to what it might be, go back and handle it again in the light of our questions, try it out, and test it in one way or another [...] Such inspection is not preset, routinized, or prescribed; it only becomes such when we already know what it is and thus can resort to a specific test…" (Blumer, 1986 pg. 44).

Systematicity can come when we already know what a thing is, and by reaching too eagerly for systematic methods we skip the problematization of what it is that we are studying. Grounded theory provides some tricks and tools for staying empirical and for navigating qualitative data, but it does not, for instance, prescribe what kinds of entities the emergent codes might describe. Given this commitment of grounded theory, I am less concerned about the issue of systematicity. The procedures of Charmazian, Straussian, or Glaserian grounded theory are not

mechanistic procedures to truth, but rather resources the researcher can use in the generation of new ways of thinking about a situation, and it is that generativity that is the ultimate goal of the methods and the methodology. The usefulness of these generated concepts becomes apparent in their application to new cases, both over the course of an investigation but also in their reuse in new contexts. The researcher can provide some scoping about problem kinds of situations where certain concepts might find lesser or greater traction.

The issue of transparency is a more important one, and there is work that can be done to make grounded theory work visible and accessible to other scholars. Making the specific procedures followed (rounds of coding, number of codes, which documents first, mechanics of code application) can be useful in demonstrating diligence and constant engagement with the data. However, given that it is the troubling and alteration of prior understandings, a generation of new understandings, that is at the heart of the method, transparency would ideally make these changes in interpretation and their connection with interactions with the data understandable to the reader. These are the what-ness of transparency for grounded theory. I have tried to do this here by recollecting important points of interpretive shift, where codes were abandoned or pushed forward in the coding process based on my own successive realizations of what this study was about, or places where rephrasing or re-relating codes accomplished an interpretive shift. While I feel that this is an important way of performing transparency for grounded theory work, it takes up a fair amount of space below, and diverges somewhat from the topics which came to make up the core argument of this study. For this reason it is hard to see how it might be done in a document which is not a dissertation. It could perhaps fit in the kind of methodological postscripts which sometimes follow books (e.g. Mills, 2000; Vertesi, 2020), but it remains an issue how this model of “transparency” might fit in a journal or conference publication.

3.2 Site Description

This study centers on the Radio Group, a research group at a large research university in the US, working in the field of reionization cosmology. Here I will describe first the larger trajectory of the field and research program in which they work, and then describe the group itself, its place and contexts of work, and the different software that it involves.

3.2.1 The 21cm signal and epoch of reionization

The Radio Group is engaged in a hunt for particularly faint emissions from the early universe. The emissions in question are referred to as the 21cm line, or just the Hydrogen line, and occur when an electron in a neutral (un-ionized) Hydrogen atom undergoes a particular kind of energy transition and emits a photon. Neutral Hydrogen made up the majority of the universe during a period known as the cosmic dark ages. The cosmic dark ages are so named both because it is the period before the formation of stars and galaxies, meaning that it was quite literally very dark, but also because so little is known about it. The photons produced by neutral Hydrogen atoms during the cosmic dark ages are still moving around in the universe, and if researchers like those of the Radio Group are able to detect them, then we will be able to see into, and derive a picture of, what the universe looked like at that time. The pressing question, and the question to which these emissions might provide the answer, is when and how stars and galaxies did begin to form. If one looks at 21cm emissions over time, early forming galaxies should appear as blank spaces in that picture, potentially allowing researchers to construct 3 dimensional tomographic maps of the development of the cosmic dawn.

The hunt for 21cm emissions is approaching the end of its second decade. Beginning in 2006, the effort has been organized around a series of large-scale interferometer experiments, which consist of arrays of antennae located in radio-quiet places in West Virginia, the Karoo desert of South Africa, and the Australian Outback. The interferometer is a particular kind of instrument which uses interference patterns between two detections of electromagnetic radiation, and they

have been used in a wide array of fields and kinds of research, including the famous Michelson-Morley experiment and in LIGO (See Collins, 2010). In radio astronomy they are used to integrate detections from a large array of radio antennae, such that the array can operate like a single instrument with an ‘aperture’ the breadth of the array. It has been in use in the field of astrophysics since the late 1940s, when it began to be taken up with a larger suite of radio astronomy techniques (Sullivan and Bertotti, 1990). These projects typically involve a large number of laboratories at a number of different universities, who work collectively to design and redesign the hardware and software of the instrument. On a specific interferometer project these efforts can last almost a decade, and a number of the professors I encountered in my observations with the Radio Group worked on prior 21cm experiments as Ph.D. students.

Across these different interferometer experiments, the overarching goal has been to increase the sensitivity of the successive instruments, and their processing pipelines, towards what is believed to be needed to detect the 21cm Hydrogen emissions. This involves characterizing and removing (or preventing) sources of interference, which include starlight and other astronomical sources, satellites, TV station broadcasts, artifacts introduced by the hardware or digitization processes, “systematics” in processing steps, ionospheric interference, and the heat of the instruments themselves. By improving some aspect of the processing pipeline, researchers can recombine an entire pipeline in order to produce a “limit”, which describes an upper limit on the power of the 21cm emission at different redshifts. In other words it asserts that the instrument has achieved a certain degree of sensitivity and it still has not detected anything that looks like the 21cm emission at a given redshift, meaning that the power of it must be lower than that. Ultimately this will begin to place limits on the theoretical models of the epoch of reionization, but as of 2025 the instrumentation has a long way to go. A recent limit in 2025, which processes data collected over a ten year timespan, has only just reached the point of ruling out more extreme (one researcher said “somewhat contrived”) models of reionization.

The process of publishing limits defines a mode of progress towards the detection of a phenomenon and the eventual ability to begin answering high profile astrophysical research questions. In the day-to-day work of the Radio Group, however, the detection is always far off, and students begin and complete their Ph.D.s without the expectation that a detection will be made soon. Dissertations, talks, and publications are made on the more immediate, and essential, work of refining the instruments' sensitivity. Any eventual detection will rely directly on decades of this kind of work. Limits are occasionally published, and marking the progress of these limits is a way of rendering progress towards a detection (Figure 1). A member of the Radio Group who shared a recent limit on a messaging platform wrote, alongside the link, "Lots of work, slow steady progress."

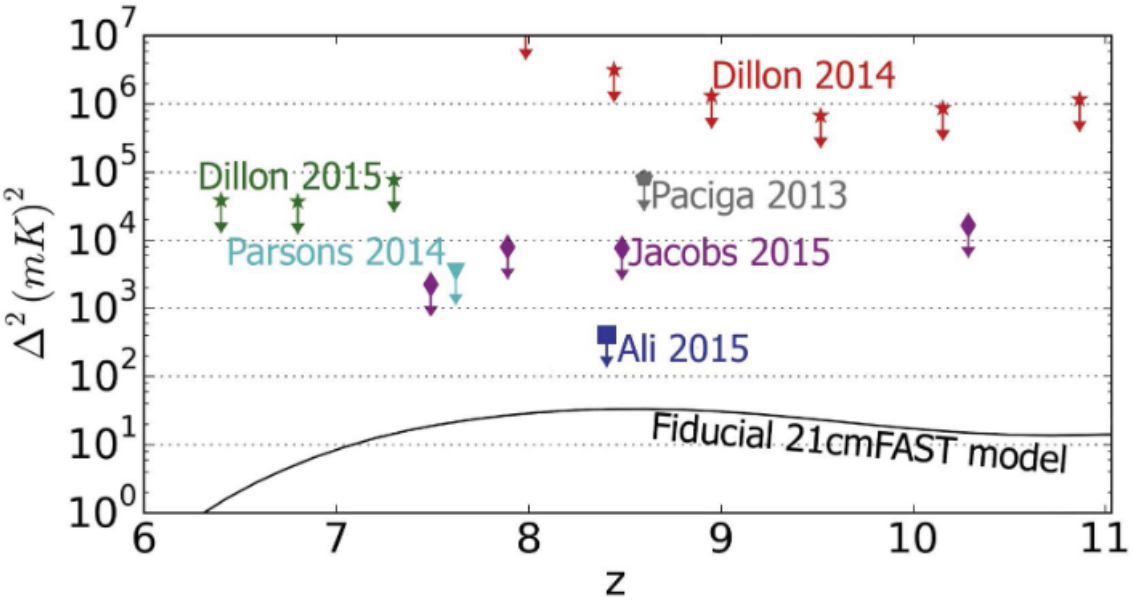

Figure 1: Plot of upper limits set on the 21cm "power spectrum" compared with a model prediction, at the bottom. Image from DeBoer et al. (2017). Note the similarity in this account of limits set on the 21cm line and Partridge's plot of limits set on the CMB in the late 60s (Figure 2).

The structure of this research program, proceeding through progressive limits on the detection of a phenomenon, is structured similarly to the earlier effort to detect the Cosmic Microwave

Background (CMB) Radiation. While the two programs are looking for different things, and address different kinds of cosmological questions, they are similar in the way progress in the research is rendered and discussed, and in the long temporalities of instrument development and theorization before an actual detection. After the initial detection of the CMB in 1962, successive attempts to measure it in detail proceeded through a series of limits (Figure 2). Similarly, in 2018, the Experiment to Detect the Global Epoch of Reionization Signature (EDGES) project reported a detection of a “global” Epoch of Reionization signal. This “global signature” is a detection of heat on a single antenna (an “average value” as one of my interlocutors put it), but it does not allow for making out any detail in the reionization signal. Furthermore, the actual detection that EDGES reported was fully twice the size EoR researchers expected. It was in fact too large to be physically possible given the current model. The EDGES detection was therefore a widely publicized discovery, but one requires confirmation and a great deal of further investigation. The report of the detection was made after the researchers involved searched for alternative explanations or instrumental effects for almost two years, and could find none (Gibney, 2018). Harvey, one of my participants, said that when you cite that detection you always have to say “awaiting confirmation.”

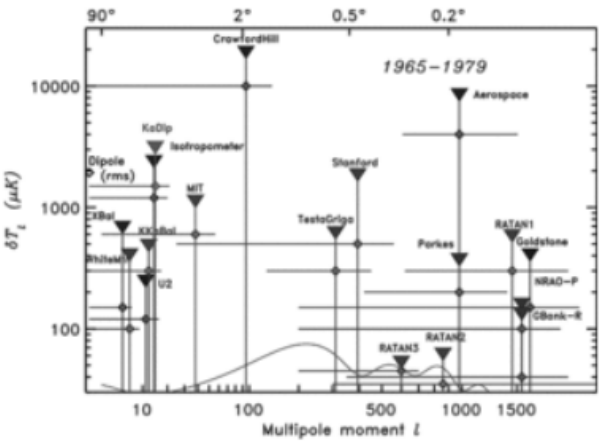

Figure 2: Upper limits set on CMB anisotropies. Image taken from Partridge (2019).

3.2.2 The Radio Group

The Radio Group itself consists of one PI, a research scientist, and 3–4 Ph.D. Students. However, their work involves collaborating with researchers at a number of other universities. This typically occurs through regular teleconferences focused on analyzing and troubleshooting different aspects of the analysis pipeline. The group's local workspace is a laboratory with benches and a small office, and it is sparsely populated by physics textbooks and signal processing hardware. However, because the group's instruments are located remotely, on the other side of the planet, the majority of their work takes place on laptops, with the group huddling around one student's computer and then another to look at plots. A large well-used whiteboard is centrally located in the room, which often becomes the venue for detailed mathematical discussions of fourier transforms, deconvolution, calibration, ionospheric effects, and so on. Students in the lab sometimes have the opportunity to travel to the telescope sites themselves. They return from these trips with stories of hardware assembled, collaborators met in person, and poisonous spiders encountered.

The more senior members of the group have worked on a couple of different reionization-related interferometer projects, and have different collaborative connections through each. At the time of my observations, they had association with an older project, the Widefield Radio Telescope (WRT), that was still running and producing data, but had also begun working on a second project, called the Cosmic Dawn Array (CDA), that was undergoing commissioning. Commissioning in this context involves performing an initial testing and troubleshooting of the hardware components and basic processing systems. CDA occupied a great deal of time of the more senior members of the Radio Group, and the Ph.D. students were split, some working on the WRT and some working on commissioning and processing for the CDA. The CDA in particular required some coordinated work between members of labs scattered across the US and internationally, which typically occurred through virtual meetings.

The group's day-to-day work consists of running and plotting data analyses, with the goal of identifying instrumental effects or "systematics" in their instrument and processing pipelines, tracking down other software or hardware bugs, or modeling physical phenomena. Much of this work is conducted through writing code, almost always in Python. This involves a huge variety of data manipulation tasks and algorithm development, but in particular it involves a great deal of plotting data, and everyone in the group was highly proficient in producing complex visualizations using Python plotting libraries.

The group builds and works with a variety of specific software packages, and my focus is somewhat uneven across these different tools. The group has a number of internal software packages that they have developed within the group in order to perform their core data analysis steps. These were written quite a few years ago in the IDL programming language, which was widely used in physics prior to its proprietary licensing costs increasing significantly. Most of the group's more recent software is written in Python. These include pycosmo, a tool developed as "glue code" to convert between formats and move data between different interferometric analysis pipelines, as well as a number of other packages for simulation and modeling. The CDA collaboration has its own array of software packages for low-level processing of the data as well as later stage analyses, and these software packages are closely dependent on pycosmo.

3.2.3 Timeline and construction of the field

In order to understand some of the findings in this study, it is important to understand the different spaces of the Radio Group and my own navigation across them. My engagement with the group began in 2019 and was preceded by a previous Ph.D. student who had worked with the group some years prior (see Paine 2016; 2017) (Figure 3), as well as my advisor's ongoing engagement with the group. This provided me with some initial rapport and understanding of my

role there.

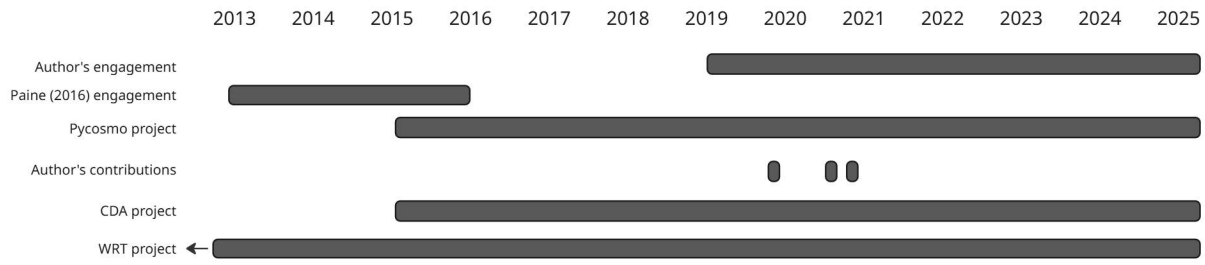

Figure 3: Timeline of the author's engagement in the field site in comparison with the large interferometer collaborations the group worked with (CDA and WRT) as well as the engagement of Paine (2016). The "author's contributions" entry indicates points at which the author contributed to the pycosmo software repository. The arrow on the WRT project indicates that it began earlier than the timeframe captured on this timeline.

My involvement with the Radio Group began with their lab meetings, which were afternoon meetings that ran as long as they had to. My initial observations at these meetings showed the particular kind of work that I would come to define as an exploratory research process, with exploratory programming as a central part of that process. These explorations involved software everywhere, but software was only discussed incidentally. The topic of discussion at these meetings was the emerging research problems of the different group members and what fresh conundrums their work had turned up in the prior week. Members of the group would gather around each student's laptop screen in turn in order to look at plots that they had produced, try to make sense of them, and to plan new tests for the next week. More senior members of the lab would interrogate students about what exact processing steps they had used or how they had calculated certain mathematical transformations. Occasionally the student would go back to their code in order to double check something, and sometimes they would make a small change and rerun it. Sometimes other members would request changes to the way the data was plotted, such as the colorbars that were used. Discussions moved from the consideration of plots on laptop screens to the whiteboard and back, and sometimes evolved into war stories about other telescopes, design decisions made long ago, or notorious bugs the more senior members had

investigated in the past. For part of my period of observation these meetings could last as long as two and a half hours.

This was the level at which I entered the exploratory process, and it differs somewhat from the level described by Kery and Myers (2017), which sits more at the interactive session level. Nevertheless, members of the Radio Group engaged in a great deal of backtracking and rewriting of their scripts from week to week, often working in the same Jupyter notebook or script file each week and simply extending it or commenting out code (temporarily sidelining particular parts of the code) in order to try something new.

After some period of engagement in the lab meetings I began attending the development meetings for pycosmo. This was a radically different space from the Radio Group's lab meetings. The development team was populated mostly by research scientists and postdocs, with one professor involved and only occasionally a Ph.D. student would join. These team members hailed from a number of different laboratories who worked with interferometers and, for the most part, on reionization cosmology. The meetings were held in the early morning, Seattle time, to accommodate different timezones, and they were strictly timebounded at one hour. These aspects created a sharp difference from the context of the lab meeting, which occurred on site (excepting the period of the COVID-19 pandemic) and consisted mostly of graduate students and occasionally undergraduates who were doing temporary research projects in the lab.

I realized that the pycosmo meetings also differed substantially in terms of process and focus. Conversations focused around planned development projects and discussing design changes, often with one member sharing their screen and programming live. Amongst these older members of the field there was a fair amount of banter over the code, but few diversions to other topics. The nature of this working session was another contrast in that in lab meetings

students would typically work on the code separately and then bring the results to the meeting, whereas in the pycosmo meetings the process was done live. It also differed in that the software was the primary object of discussion and ‘scientific’ topics emerged in an ancillary way, as use cases to consider in a particular design decision or interesting stories behind particular problems. Sometimes members of the team shared development tricks they had recently encountered or discussed the potential usefulness of a new development tool they had become aware of.

I also attended a smattering of other events that the group was involved with, such as a tutorial session amongst the students on cloud computing services (see Sutherland, Paine, and Lee, 2024), departmental reading group sessions focused on the Radio Group’s work, and dissertation defenses of members of the lab.

The choice to begin observations with the pycosmo development project was an intentional choice. My involvement with the topic of scientific software preceded my involvement with the Radio Group, beginning with prior work observing in software carpentry-style workshops for researchers (Feinberg et al., 2020). In the group’s lab meetings I found a space where software was tangled up in an unpredictable and emergent research process, but I knew from statements made by others that the development of pycosmo was quite different from this. As will be discussed in my findings, pycosmo’s development patterns were atypical for both for the CDA but also in general discussions about scientific software. Pycosmo seemed to flout all of the typical associations with scientific software (Heaton and Carver, 2015). In this way it constituted a critical site to contrast with my observations of the lab meetings and also to explore the realized possibilities of robust end user development.

In this sense engaging the Pycosmo project was a kind of theoretical sampling, which allowed me to tie the *data collection* process to the specific dynamics of my developing theory. This is in

fact a point on which Vaughan (1992) finds some affinity between Grounded Theory Methodology and case study methodology, in that selecting cases for intentional difference has the potential to present the researcher with surprises that prompt or force theory elaboration. This of course also delimited my study in various ways (Cohoon and Howison, 2023), focusing it on the core of what I call here *production code or collaboration code,* and left out other activities, such as low level processing development and simulation work.

My involvement with both of these spaces was usually in a ‘fly on the wall’ mode of observation, but occasionally I would become involved in lab meeting discussions whenever the topics turned to ‘social’ issues of the collaboration or the long term dynamics of the interferometer projects that the group was involved in. My prompting of these topics seemed to create a context for the discussion of these kinds of issues that did not frequently come up in the regular lab meeting, focused as it usually was on ongoing research problems. On one occasion I gave a talk to a cosmology-focused reading group in the department (the same mentioned above), which gave a broader context for addressing some of these issues. During my observations with the pycosmo group, I also made three small contributions to the codebase, which became important sources of personal experience with the group’s development and contribution processes.

The overall shape of my engagement, which centered on the Radio Group and extended out along their collaborative engagements, was a construct both of my intentional ‘sampling’ choices but also of the constraints of the field (Parmiggiani 2017; Karasti and Blomberg, 2018). In some cases I was limited by the number of meetings I was able to attend, but my choices about which contexts of software-related work to study came with presences and absences of certain kinds of scientific work. This was particularly clear in relation to the prior work of Drew Paine, which took place during the group’s engagement with a different interferometer project and therefore involved a slightly different coordinative context (Paine, 2016).

3.3 Data collection and analysis

My analysis was based on a couple of related sources of data: 1) observations in lab meetings and development meetings and the field notes that they generated, 2) interviews with members of the Radio Group and their collaborators, 3) documentary traces from Github of issues and pull requests made against specific software repositories, and 4) other documentary evidence such as memos written by members of the group, scientific publications, computational notebooks, and various other web pages and repositories. The first three (field notes, interviews, and documentary traces from Github) were used in the qualitative coding process, while the others were used to inform arguments, interview protocols, and vignettes.

Data source	Count
Interviews	21
Ph.D. student	8
Postdoc/research scientist	6
Professor	7
Field notes	45
Github Artifacts	46
Total	112

Table 1: Summary of coding materials.

Observations were conducted as described in section 3.2.3, and I wrote field notes describing my reflections and interactions during these observations. A subset of these field notes were selected for coding based on their completeness. Interviews were conducted both in person and remotely and were recorded and transcribed using a transcription service. Documentary evidence from Github was collected by downloading issues and pull requests (PRs) as PDFs which could be coded. All participants, projects, and software tools have been pseudonymized.

3.3.1 Coding and analysis

The coding process proceeded from initial coding through focused coding and memo writing (Charmaz, 2006; Glaser and Strauss, 1967). I coded the field notes first, and then the interviews, coding at the level of sentences or paragraphs. I coded liberally, not worrying about semantic overlap or building more general concepts. At the end I consolidated a few clear duplicates in codes (such as plural versions of a code) and deleted some that were quite distant from my research question. I clustered the codes into unnamed groups based on their relevance to each other, and wrote descriptions of the groups as a whole. This process was a preliminary round of focused coding: it forced me to attempt to define what these sets of codes were about, and how they might connect with my larger research questions. I wrote the descriptions based on re-examinations of instances of the codes and comparison across codes, and it resulted in moving codes between clusters or re-terming them. This process resulted in 143 codes.

I then used this initial set of codes and the clusters to guide my collection of artifacts from Github. I identified a set of software repositories that the members of the Radio Group worked on based on my observations, and examined both their pull requests and issues page by page, starting with the oldest entries. The goal of using these PRs and issues was to examine discourse over the activities of contributing code and negotiating contributions rather than for examination of the code contributed. For this reason I made selections based on a couple criteria: 1) they had at least one substantial text interaction between developers on the project, and 2) the contribution being made or the discussion on the issue had some relevance to one of the categories I had developed in my first round of coding.

I then engaged in focused coding on my original set of 143 codes. This involved comparing codes and looking across instances of a given code to prompt the question of what it was about. There were a couple of major shifts in focus that occurred here. In particular, my initial coding

process had produced a large number of codes relating to many of the typical themes of cyberinfrastructure: standardization, interoperation, balancing different interests in design, and so on. I realized that my interest in these issues was to draw connections between these efforts and those of exploratory work. In other words, what I felt needed to be understood about software engineering practice in the sciences was tensions that emerge between movements towards robustness and infrastructure and local activities of exploration as a process. This itself bears a great deal of continuity with studies of cyberinfrastructure, but it is not focused on the standardizing and interoperationalizing activities themselves. In some ways it is focused on their counterpoint in exploration.

I also made the decision to pursue a notion of capacities at this point, because the concept encompasses concerns about phenomena and concerns about software. This emerged from an alternative casing (Tavory and Timmermans, 2014) of scientific work under the view of science and technology studies approaches and software engineering approaches. Switching between these two views I seemed able to talk about the iterative change of software or the iterative change of phenomena, but I realized that I needed a way to talk about them together. This became particularly clear in plotting a diagram of the larger trajectory of the RFI research program, for which I could not decide whether the entities depicted were pieces of software or something else. The notion of capacities captures this middle ground, and I adopted it at this point, drawing it in part from Rheinberger (1992), but using it for this specific purpose.

Another point of consolidation was taking a number of emic terms for characteristics of software and subsuming them under a ‘software epithets’ code. The emic term “hacky” was initially important for me to capture, but it turned out to be one among a number of terms (such as “sloppy”, “clunky”, and “clean”) that were used to characterize and give moral valence to software produced in different ways. I realized in the second round of coding that my interest was not in outlining all of the specific associations of these different terms, but rather to look at

how these characterizations were resources for the collective distinction of research code and collaboration code. Describing epithets in general highlights what these descriptions do vis-a-vis a larger activity of distinguishing types and circumstances of code (distinguishing research code and collaboration code), and the attribution of a moral valence to these different types and circumstances. In other words rather than listing the names applied I wanted to capture what the act of naming does for an effort to change practice in the collaboration. This could only become visible once I had developed that larger strategy of changing practice from elsewhere in the data. In this way the code took shape together in the focused coding process.

I removed a number of codes around the notion of users. I realized that while the inclusion of more stakeholders was an essential part of the notion of collaboration code (it's in the term, after all), I a) did not want to focus in a detailed way on the rendering of different stakeholder needs, but rather on general techniques for negotiating them, and b) the framing of "users" was distracting from local understandings, even if it was not an entirely etic concept.

During the first round of coding I also began memoing, particularly on narratives or events that were significant or on particular early codes. These memos were as Lempert (2007) describes, focused on analysis rather than formality, describing and redescribing the codes in order to work through my own interpretation, rather than for detailing theory in a way that would be coherent to others. During the focused coding process I wrote further memos aimed at presenting the concepts to others, and I drew diagrams attempting to outline their interrelationships.

After the process of focused coding I had a list of 42 codes, which I used in recoding the data, including the Github issues I had collected. This produced a few new codes, including 'mnemonic artifacts', which quickly swallowed a number of other issues that had previously seemed only tenuously connected to my primary interests. This final process of consolidation and reshaping resulted in a final set of 29 codes (appendix A).

3.3.2 Research Questions

At the outset my investigation was sensitized by ongoing discourses around scientific software, but evolved to focus on specific dynamics that were clearly present in my case and absent in the broader discourse. The ‘problem’ of scientific software was both an etic concept, which I encountered in the literature on software engineering, but also an emic concept discussed by members of the Radio Group and the CDA. It is one of the primary goals of grounded theory methodology to change the researcher’s understanding of an issue or situation, and this was reflected in shifts in my research questions over the course of my observations, coding, and writing (see also Charmaz, 2014, pg. 77). These shifts reflected points where engagement with empirical content resulted in changed understandings that I could use to better frame the object of my investigation and both prompt and refine further analysis.

Early on in my observations with the Radio Group and the Pycosmo development meetings I realized that there was a great deal of work that was not well accounted for by the production-focused conceptualizations in the literature, and that it would be important to characterize how the group proceeds and organizes their work in situations where they do not have clear requirements or “oracles.” This produced a first set of questions:

- *(RQ1) How do members of these two research software development efforts co-construct research problems and research software requirements?*
- *(RQ2) How do members of these two research software development efforts manage change in research problems and requirements as projects develop and stakeholders come and go?*
- *(RQ3) How are “best practices” and techniques associated with software engineering leveraged in these development processes?*

In relation to the literature’s focus on the difficulty of obtaining requirements in the sciences, the first question focused my attention directly on the process by which requirements take shape, and uses the notion of problem formation as a way into that process. This was intended to

further my investigation of a dissonance I had seen between the literature's focus on requirements and the fact that the Radio Group pursued apparently systematic processes of software development and use largely without them. The second question also reflected the observations I had made that uncertainty in what software should do came both from the changing understanding of a research problem as well as from a changing set of stakeholders. Finally, the last question directly addressed what I saw as a central gap in the literature that I had not yet closely examined in my case, which was how software engineering techniques were actually taken up and integrated into a scientific work environment.

This first set of questions was originally developed through observations with the Radio Group, but at the outset of the study there were actually intended to be addressed across the Radio Group and an additional case, which was also in the field of astronomy and astrophysics but involved software engineers working on the software projects. Initially I proposed this second site as a complement to the Radio Group, and when this comparative effort later became infeasible, little conceptual reframing had to be performed because the questions had been formed around the Radio Group. These initial questions did serve to guide early interviews and questions and probes during observations.

These questions also served to guide the initial coding process, but they were also reshaped through conceptual development in that process. This produced three slightly different versions of my questions:

- *(RQ1) How do members of the Radio Group leverage software in the process of problem formation?*
- *(RQ2) How does the Radio Group maintain software as research problems and stakeholders change?*
- *(RQ3) How does the Radio Group integrate software engineering practices into their research work?*

Examining the successive process of problem formation did turn out to be a fruitful empirical object, but I needed to better frame my question around how software development and use occurred prior to the emergence of requirements, and not just how requirements emerged. In part this was because it became clear that there was a great deal more to be examined about this process than just the formation of requirements. These were the rationales behind the change in research question 1. Research questions 2 and 3 shifted only slightly, reflecting some changes in terminology, for instance from “best practices” to software engineering practices.

The last shift in my research questions came after the focused coding process and during writing, when I needed to bring the outcomes of my investigation back into conversation with the literature. At that point I was able to return to my initial curiosities with new terminology and concepts developed in the focused coding process as well as with some shift or extension of interest. For instance, I picked up the term exploration as a larger encompassing framing for the activities I had observed in the Radio Group that I had initially approached through the process of problem formation. While problem formation remained central to my analysis and my methodological approach, I now had the term exploration to describe a larger process in which it took place. I also now had a clearer conceptualization of the tensions that were at stake in the organization of a research software system, and my concern was more clearly focused around the issue of unplannability, allowing me to be more specific about the aspects of software engineering practices (long-term planning and design processes) that mattered to this study. This resulted in my final set of research questions:

- (RQ1) *How does software become a tool for exploratory work?*
- (RQ2) *How do researchers reconcile flexibility and rigidity in their software tools?*
- (RQ3) *How does a research group incorporate software planning and design processes into their research work?*

These questions served to prompt and shape the conceptual development that occurred in the final writing process, and particularly the connections that were developed in that process between my own findings and relevant concepts in the literature.

3.4 Terms

Based on the literature laid out above, and the methodological context provided in this section, there are a number of terms worth clarifying.

The Radio Group themselves switch between vague and extremely specific language depending on the context, and here I have chosen a middle ground between these things. For instance, interferometers of the type considered here are most accurately described as arrays, but members of the project frequently refer to them simply as “the telescope”, and I will do that as well. The group has highly specific language to describe the dipole antennas elements which do the actual collection of data, but for my purposes I will refer to them as “antennae.” Most of the interferometer projects are termed as “experiments”, which distinguish them in important ways from the concept of an observatory, but here I will usually refer to them in a more general way as “projects”, given that my interest is in particular on the work and organization of these projects over time. The group works in a larger field of astrophysics, although they sit in the physics department, and identify in a number of ways with the field of physics. I will refer to their work at the high level as cosmology, within a larger field of astrophysics.

Rheinberger uses the terms “experimental system” and “experimental situation”, but my notion of a research software system is somewhat broader than this, comprising not just the objects that researchers manipulate but also the processes of work and coordination through which differential reproduction plays out. I will use the terms “instrumental system” to mean something similar to Rheinberger’s “experimental system.” The notion of the experiment has particular connotations in the philosophy of science that diverge quite strongly from the kind of data

analytic work that the Radio Group pursues, if not from the larger framing of their interferometer "experiments." An experiment in classical understandings is a justifying activity, usually with a clear hypothesis (see Steinle, 1997), and it relies on intervention on the scientists' object of investigation. Rheinberger's work takes place in the heyday of experimental biology (see also Strasser, 2019), and he also takes a very different meaning of the activity than in these understandings of experiment. For my purposes, however, the term "instrumental system" helps avoid discussions about the status of data analysis or simulation and modeling as experiments (Tamborini, 2020), which I do not intend to weigh in on.

A major aspect of this dissertation is a set of practices which are understood to be established in a sphere of industrial software development which are often recommended to scientists as minimal and basic practices that they can adopt to improve their work. These are often referred to as "best practices" and they vary somewhat in their content. Here I will follow Heaton and Carver (2015) in calling these collectively "software engineering practices."

Chapter 4:
Exploration and the development of problems

The first thing that is necessary to understand the research software system is to make exploratory programming visible as an aspect of scientific work. Kery and Myers' (2017) account of exploratory programming provides some essential aspects of the activity, including that it involves tradeoffs in code quality, frequent backtracking, the retention of unused code options, and difficulty in collaborative work due to the "informal" practices frequently used. Here I will look at a couple more characteristics that fit this kind of exploratory programming work into a larger process of exploration in research work. First, exploratory work is organized around epistemic objects and problems rather than requirements. While problems do not provide detailed specification of what software should do or what outcomes it should produce, they do provide a basis or framework for designing new trials and tests. Problems are constituted by a relationship between a researcher and some experimental apparatus, and often they are centered around anomalies, things which are determined to be "out of line" (Star and Gerson, 1987) with regard to some particular operations of a system. Such anomalies are epistemic objects (Rheinberger, 1992a; Ewenstein and Whyte, 2009): they are characteristically incomplete or only vaguely apprehended by the researcher. Precisely because of their incompleteness they precipitate their own further development or unfolding. It is through interaction with these incomplete objects that researchers organize and structure their work.

Additionally, exploratory work involves engaging these anomalies through open-ended tests, which differ from most tests typically involved in a software engineering process. Exploratory or open-ended tests are run in order to see what happens. They do not have specified requirements against which the output will be evaluated, but rather the output is intended as a point of departure for a sensemaking process (Weick, 1995). This defines a particular

orientation towards running software, which involves among other things a great deal of retrospection on what happened in a given test.

Although this exploratory process is inherently unpredictable, it is nevertheless structured in a number of ways. It is important to clarify, first, that researchers do engage in many kinds of planning, from planning new tests for the next week to the multi-decade project of detecting the 21cm line. However, unplannability as it is meant in both the science studies literature and the software engineering literature does pervade their work. Specifically they cannot draw on any clear script or blueprint for designing their software or carrying out their interactions with nascent phenomena. Nevertheless, researchers are able to structure or systematize their work in a number of ways. Exploratory tests do not have predefined outcomes, but they are shaped by established practices, techniques, or repertoires (Ankeny and Leonelli, 2016). Moreover, anomalies are “managed” (Star and Gerson, 1987) through an iterative process of reproduction that is routinized, but nonetheless generative of new interactional possibilities. Lastly, although researchers do not know where their research projects will take them or how long they might play out, they can structure their investigation within a regular rhythm of lab meetings and development work.

In order to demonstrate these aspects of exploratory work, I follow a couple examples of *problem formation* in the Radio Group. These are research projects which unfold from the appearance of an anomaly and research problem, through a process of reproducing and transforming the anomaly and ‘the problem’, towards points of relative closure where what were previously anomalous interactions of the group’s instrumentation are turned into regularized or productionized software tools. In particular my analysis focuses on techniques for the detection of radio frequency interference (RFI). This is one strand of research that the group pursues, but it is one amongst a number of others. For the sake of simplicity and continuity across examples, I have scoped the discourse to this part of the research.

Section 4.1 will trace the larger process of generative reproduction that is at the center of my account of exploratory work. It also demonstrates how this activity is organized around anomalies and problems rather than plans or requirements. Section 4.2 will look more closely at the process of exploratory testing, and how different components of a larger research software system are problematized and re-problematized in the research process. Section 4.3 will look at how this exploratory process is organized in time, how it comes to take on a particular rhythm despite its uncertain direction and time frame. Lastly, in section 4.4 I will summarize some of these characteristics of exploratory work and its implications for our understanding of scientific software. Throughout these sections, it will become apparent that understanding how the Radio Group's exploratory work is accomplished requires understanding also their processes for productionizing software. This is where I will turn in the next Chapter.

4.1 The generative reproduction of anomalies

A primary characteristic of exploratory programming is that it is centered more around problems and anomalies than requirements or plans. As has been outlined by others, anomalies are a ubiquitous part of scientific work, and the day-to-day activities of research involve the manipulation and management of anomalies (Star and Gerson, 1987)[1]. The argument here is that anomalies and the problems that emerge around them are the sites of growth or change in an instrumental system. Anomalies are places where a system escapes or diverges from a prior state of working order. That is, where it does something new. They are not only new, they are also not entirely definable or explainable. In this sense they are epistemic objects (Rheinberger, 1992a; Ewenstein and Whyte, 2009): they are characteristically incomplete, and they prompt further development and unfolding precisely because of this incompleteness. Anomalies are

[1] Imre Lakatos used this point to argue against Popper's understanding of falsificationism, pointing out that any trivial refutation of a theory is not necessarily an empirical failure because in practice "research programmes grow in a permanent ocean of anomalies" (Lakatos, 1989, pg. 6).

also typically *problematic*. While some anomalies might simply be ignored, many obstruct or call into question the larger project of the researcher. In this way, like a breakdown (Star and Bowker, 2006), an anomaly prompts (or demands) reflection on a situation and potentialities, as well as active efforts to understand 'the problem.'

It is through interaction with these epistemic objects that researchers come to organize their work without the ability to define stringent plans or specifications. While epistemic objects are not scripts (Schmidt, 1997) and they cannot define clear next steps, they can nevertheless help shape the trajectory of an investigation through the generation of questions, curiosities, and problems. In this sense problems and anomalies in fact bear some similarity to plans and requirements in the specific sense that they can be resources for the projection of future action, although they are used in quite different ways. Another way that this could be framed is that in the absence of detailed plans or requirements anomalies and vaguely defined problems are what researchers have to work with. While this is a fair characterization, it is important to note that researchers actively seek out those parts of their system that are anomalous or not yet understood. As I will argue here, anomalies or research problems are points of departure for new research projects, and as such researchers operate around them more as resources than as obstacles.

Here I will describe how the Radio Group organizes their work around anomalies and problems, specifically through a process of producing and reproducing anomalies. Where the typical notion of reproduction or replication is intended to mean doing the same thing again, what I mean here is a process of re-enacting a previously observed anomaly in a new way. In this sense it is not a process of testing or demonstrating established understandings, but rather a process of probing the interactional possibilities of a system. In this meaning reproduction is a process of carefully and strategically doing new things (see also Feinberg et al., 2020).

I will examine this process through the work of Noah, a graduate student in the Radio Group. Noah's work took up anomalies that had appeared in a prior student's research and used them as a jumping off point to refine the group's ability to detect radio frequency interference (RFI). RFI refers to any kind of signals, often but not always produced by humans, which might be detected by their antenna array and therefore interfere with their ability to detect their signal of interest, the 21cm signal. All of the interferometers (antenna arrays) that they work with are placed in very radio-quiet parts of the Earth, but nevertheless they can still pick up signals from distant TV broadcast stations, strange ionospheric interactions, airplanes flying along the horizon, and satellite constellations moving overhead. All of these signals can interfere with or "contaminate" the data in the sense that they can either block out a potential detection of the 21cm line or introduce confounding signals that could be mistaken in analysis. For this reason they need to be detected, identified, and removed from the data, a process the group referred to as "flagging."

Noah started his investigation with a problem that was left over from the dissertation work of a prior student, Mason. Mason had developed some Python code called cosmo-rfi which would search the group's data looking in particular for kinds of RFI that were flickering or moving. Part of the algorithm that Mason wrote involved performing time-based subtractions which would remove features of the data that were not changing rapidly (most stars and galaxies, for instance) and leave things that were likely sources of radio interference. Mason has successfully detected a number of new kinds of interference with this technique. He was able to demonstrate, for instance, that TV broadcasts were reflecting off of airplanes moving along the horizon and getting caught by the antenna array. His software was able to identify these and many other features in the data and "flag" them, such that future analysis could avoid that interference in the data.

However, in completing his dissertation analysis he had also been able to demonstrate that there was still some kind of interference he was not catching. Some portions of the data he was using showed clear interference even though his flagging software did not detect anything in those segments of data. This appeared as “excess power” in the final analysis results Mason had produced. This excess power was an anomaly in the sense that it could not be properly explained, categorized, or managed within the present working order of Mason’s RFI detection software. When run, cosmo-rfi produces an outcome which is outside of what members presently understand about RFI and its interaction with their detection software. Mason’s work had generated “answers” about previously unknown RFI, but it also generated new anomalies and new problems.

It is worth reflecting on how these kinds of research problems are managed as resources within the group. Problems such as Mason’s “extra power” might be recorded in publications (especially dissertations) or “memos” created by researchers who found them. Importantly they were remembered by more senior members of the group, especially by Magnus, the head of the group. They were then given to new students as the starting point for new projects. As Magnus put it, “the good news is we have no shortage of problems.” As he went on to explain, what defined the research project that a student took on was an “admixture” of concerns, including the landscape of available problems (of varying urgency), the strengths and skillsets of students, and what kind of “portfolio” of work they would need, which depended on what they were hoping to do after graduation. Moreover, Magnus described how in his experience large experimental physics collaborations often had some negotiation over which topics would be taken on by which students, especially for dissertation projects:

> “But if they want to do a more academic path and postdocs or things like that, or go for prize fellowships, they want to have really meaty science in their thesis, but in order to get that topic, they have to have done something really useful for the collaboration. And all of this is behind the scenes and kind of quiet and kind of, I mean, I try and be explicit

> with the students, but there's this little quiet horse trading that goes on…" (Magnus, professor).

Often new students would do "something useful" for the collaboration in the early part of their Ph.D. and their ability to work on a prestigious or "meaty" scientific topic for their dissertation work was dependent to some degree on this kind of service work that they would perform for the collaboration. This service work might include chasing down more minor anomalies or building out new computing resources, such as setting up cloud computing services (Sutherland et al., 2024). More meaty science meant pulling the group's entire analysis infrastructure together to perform a fuller analysis of the data, and to mark down a new overall sensitivity of the instrumentation towards a possible detection of the 21cm signal.

My goal here is to demonstrate how problems are used to organize and delegate future investigations. Problems are starting points that can be maintained and delegated amongst researchers. They might have more or less prestige, or fit better or worse with different kinds of skillsets. They also provide direction and shape to nascent investigations. Mason's excess power anomaly, for instance, appears in data products that the cosmo-rfi software has designated as clean. This is a vague indication of a problem but it does have shape. It indicates interference in the data, and it indicates that this interference is something other than the particular kinds of features they have already looked for: it is *something else* from previous phenomena. This 'else-ness' is not much to go on but it is a lead, a resource with which to shape new tests. As we will see, Noah is able to transform this anomaly and develop its characteristics. This in turn helps shape new tests.

Noah first processed the data to a further point in the group's analysis pipeline in order to produce actual images of the data projected on the sky as it might be seen from the array's perspective (Figure 4). Mason's approach had essentially looked for RFI across the whole sky, but through the imaging approach Noah might be able to locate spatially where this new kind of

RFI was emerging on the sky. This would allow him to develop more targeted statistical tests, which would have a better chance of identifying it. Noah leveraged the group's primary analysis software, a very large and complicated piece of software written in IDL, in order to process the data to the point where an image could be constructed. He then wrote some code in Python to plot the outputs of that processing "on the sky." This produced a circular view of sources on the sky, and in certain circumstances Noah was able to see some effects that looked like RFI (Figure 4). In particular this included long strands of black dots running towards the rim of the plot, the horizon, where the telescope should not even be particularly responsive.

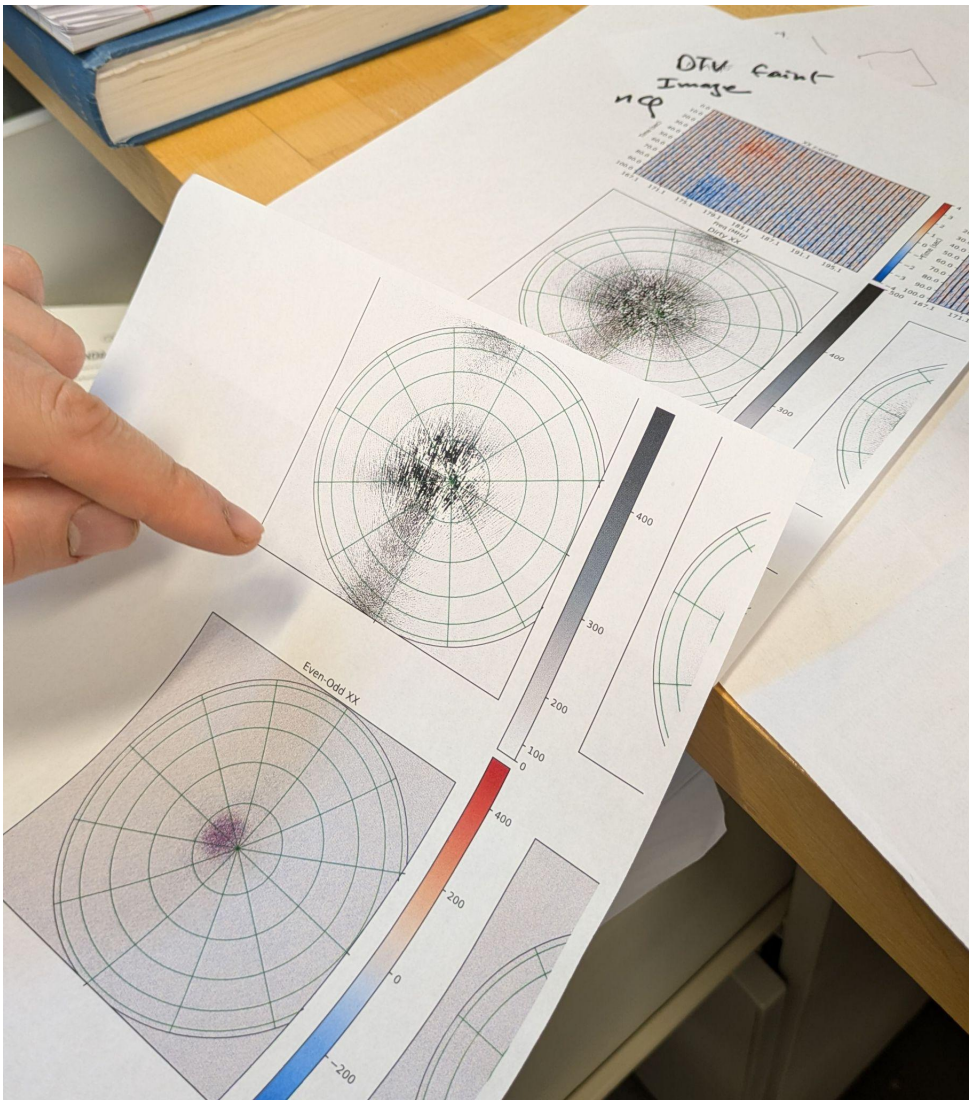

Figure 4: Plots of data “on the sky”, with the circular region of the plot representing a view upwards from the point of view of the telescope, with the rim of the plot representing the horizon. Noah points to a long streak of data points indicated in black, which is likely produced by RFI contamination.

While this “imaging” approach did show the effects of RFI in certain circumstances, it also had a number of drawbacks. It was not a viable approach in the long term because it was computationally “wildly expensive” as Magnus, the head of the group, put it. This was especially problematic because sections of the data would simply be thrown out after the processing had been applied. In this sense the imaging approach was not a very “do-able” (Fujimura, 1987) way of framing the problem of RFI detection in the long term. More importantly, however, they were not able to get detailed detections of the RFI they were looking for:

> “…so we tried imaging. It didn't work very well. Imaging the UV plane was, there's a bit of that that was, maybe there's a way of finding the signal there, but a lot of it was ‘why wasn't the imaging working?’ Well, to be honest, I still don't know why the imaging wasn't working as well as it is. And we talked about looping back and looking into that sort of thing” (Magnus, Professor)

Noah was able to see some RFI in certain cases, but he was not finding very much RFI that was not *also* caught by Mason’s earlier algorithm. This definition in the negative was all the group had of their present object of research, and the imaging approach had given them no new handle on that object.

At this point the group heard from collaborators at another university that an RFI detection technique they were developing was seeing traces of RFI in observations (segments of collected data) where cosmo-rfi had not seen any. Importantly, their approach was also a full sky approach, suggesting that it was possible to find this new RFI without trying to localize it spatially. By looking at the approach that this other group had developed, members of the Radio Group got the idea that perhaps this RFI they were looking for was not caught by cosmo-rfi because it was not moving or changing quickly. They knew that the cosmo-rfi algorithm would

decrease the “power” or strength of RFI features in the data overall, and so it might be systematically overlooking features that were not moving in space or changing rapidly.

This idea led Noah to a new approach. He started working with the “visibilities”, lower level data products that Mason had also used in his detection strategy, but developed a new detection algorithm. Where Mason’s “sky subtraction” method was optimized for finding rapidly changing features, Noah developed a new algorithm, or “statistic”, focused on identifying slowly changing features in the data. Running this new system against the data produced a number of new phenomena, including a feature that the group called the “evil cow” (Figure 5) and one that they called the “stratospheric clouds” (Figure 5), among 4 or 5 others. The evil cow was so called because it consisted of a sinister pattern of red and blue in the blob-like shapes of a cow’s spots.

These new phenomena represented a major breakthrough in the investigation because they concretely transformed the problem that the group was encountering. Initially, the anomaly was only understood as some kind of interference not caught by an algorithm looking for rapidly changing or moving sources. The group then developed the idea that they were looking for RFI that was not changing rapidly in time, which gives some further potential shape to their object of investigation. With Noah’s new algorithm, the anomaly splits into many, and takes on concrete shape and character in the dimensions of time, frequency, and power. The evil cow, for instance, tended to appear in certain frequency ranges associated with digital television broadcasts, it tended to occur over long time periods, and it exhibited changes in power that varied by frequency. This was not conclusive about the nature of this phenomenon, but the character of the group’s anomaly, and the character of their problem, had been transformed significantly.

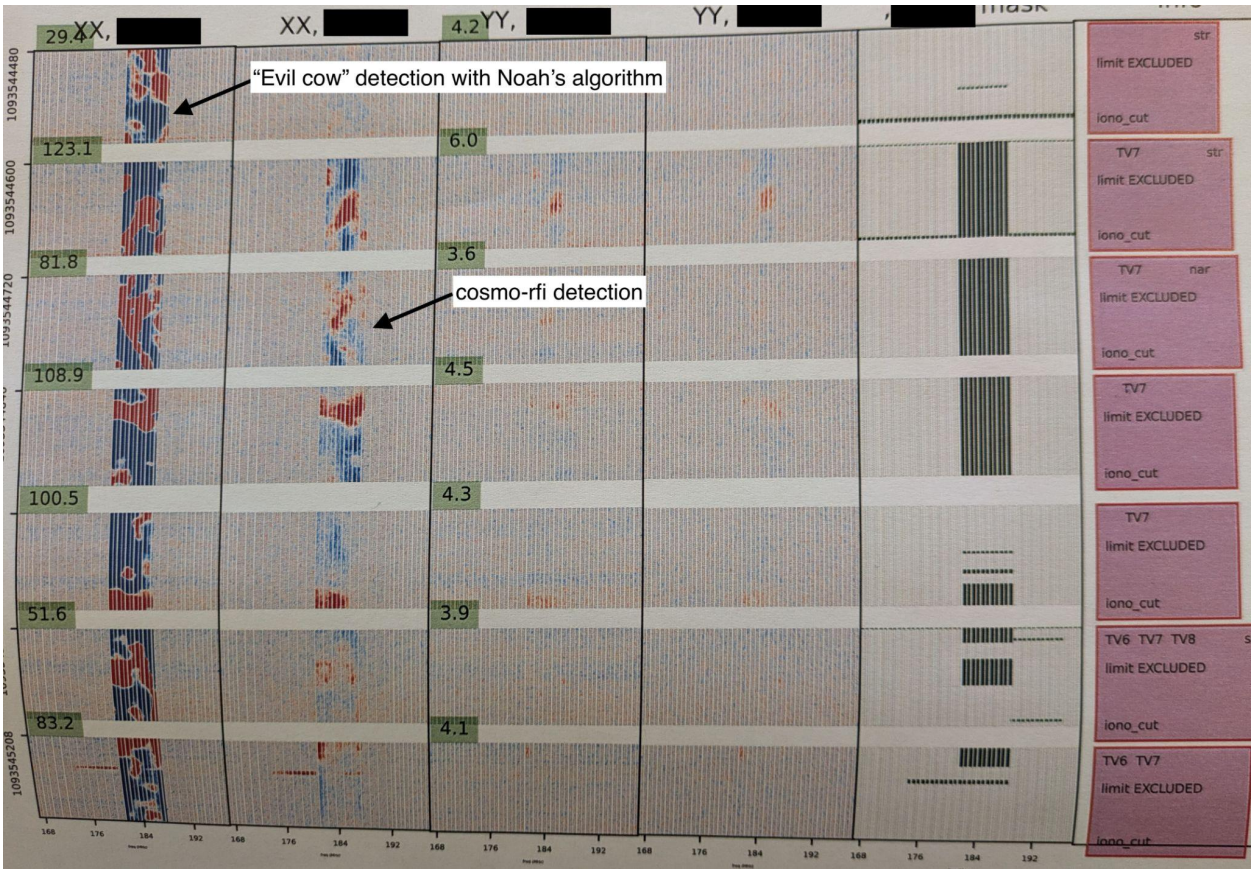

Figure 5: A plot produced by Noah's software, showing data collected from the Group's array by time on the y axis and frequency on the x axis. Columns indicate different analysis algorithms. The most recent iteration on the left clearly shows the phenomenon the group calls the "evil cow", while the second from the left shows the same phenomenon appearing faintly in an earlier iteration of the algorithm.

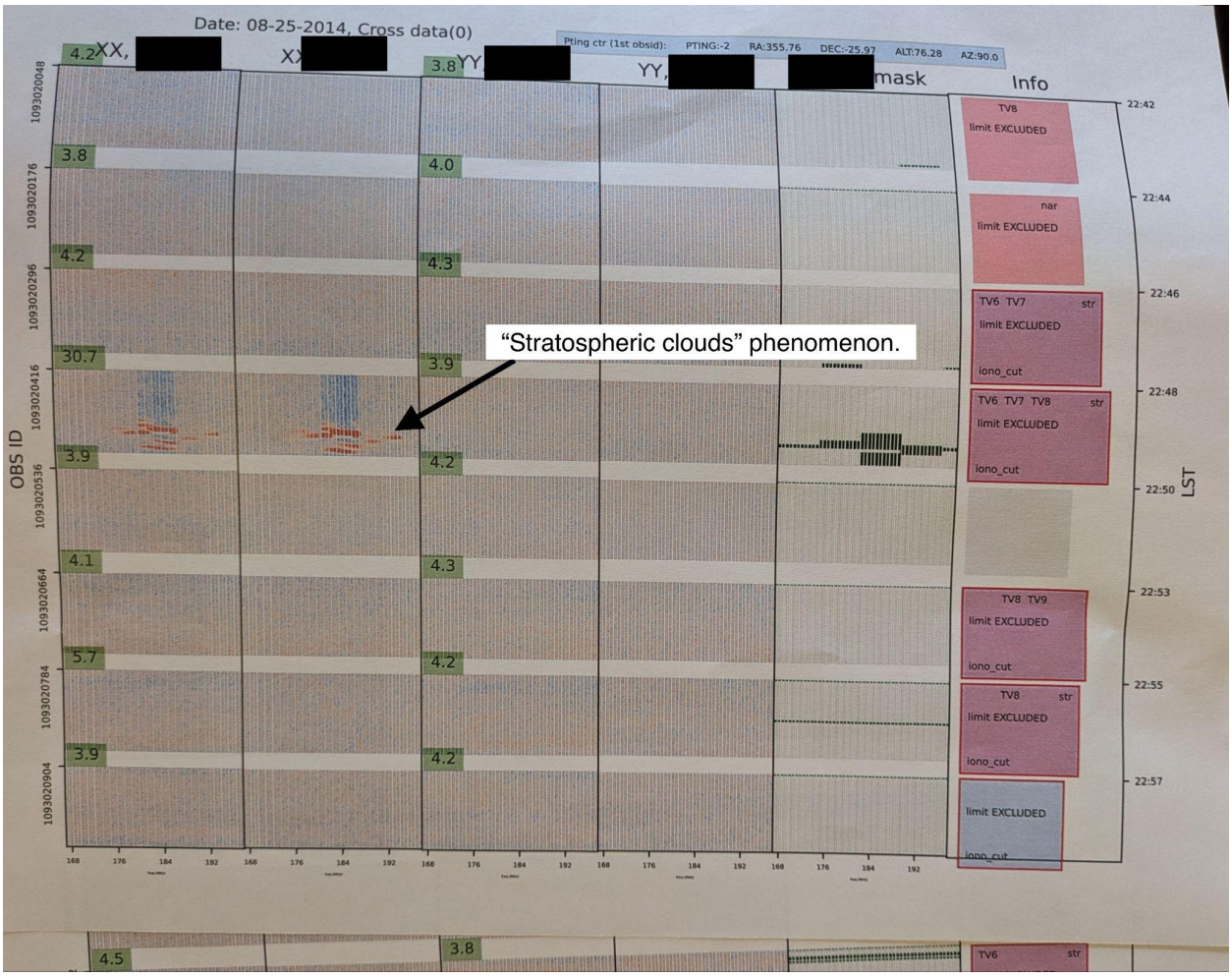

Figure 6: A plot similar to figure 5, showing time on the y axis and frequency on the x axis. This section of data produced the "stratospheric clouds" phenomenon, so called because of its unevenly distributed horizontal lines.

This progression of developing a vague and poorly understood anomaly into a more concrete, comprehensible phenomenon is the basic pattern of the Radio Group's research work. Through iterative software enactments members of the Radio Group can transform the anomaly itself as well as the 'the problem.' In Noah's work this is a move from a vague problem of "excess power" in observations that were supposedly clean, to a more specific problem of the "evil cow", which appears in particular ways and suggests new avenues of investigation. Moreover, the problem has multiplied, producing not only the evil cow but also a number of other phenomena. These

other anomalies were out of scope for Noah's dissertation work, and would have to be picked up by future students as points of departure for new investigations.

In this process they are not working from plans or requirements, as there is little understanding of what the software should do, or what outcomes or phenomena should emerge. However, they can look retrospectively on past enactments of the phenomenon in order to design new tests. This is what was done with Mason's initial excess power anomaly, and the evil cow is, at the time of writing, shaping new tests that Noah is pursuing. This is a more general pattern of the group's interaction with anomalies. For instance, if a feature in the data changes based on directionality, it may prompt the group to look further for a phenomenon that might be occurring on site, in the area around the array. If it does not have directionality in the data they may go hunting for a bug in the software or hardware processing system. These are somewhat simplistic examples, and it is often a complex process to figure out exactly what next steps might be warranted by the appearance of a given phenomenon.

It is in this way that anomalies function as epistemic objects. In their incompleteness they prompt, and provide some direction for, new investigations or tests. As Rheinberger (1992a) puts it, they are question generating machines. The Radio Group's assemblage of software resources not only constitutes these objects, but also provides the basis for interacting and engaging with them. It is through a kind of "conversation with the materials" (Schön, 1983) that anomalies take on new shape and researchers change their understanding of the problem.

What is important to recognize here is that members of the Radio Group spend a lot of time with these kinds of anomalous objects. Provisional, incomplete objects, such as the "evil cow" take up the lion's share of the Radio Group's efforts and concern. These kinds of nicknames were extremely common in the Radio Group, and often projects or long-running problems were referred to this way, including "worms", the "one third line", "stratospheric clouds", the "icicle",

and so on. Paine (2016) further describes investigations of the “fourth line bug.” Especially in the study of RFI, these names often reflected a particular appearance of the phenomenon in one or another kind of plot. In other cases they had less to do with the appearance of the phenomenon and simply operated as handles to distinguish them from one another. One previous student had been working on features that appeared in different “flavors” of cable used on site at the array, and so she began to refer to the features using flavors of icecream, such as “rocky road.” All of these nicknames reflect the status of these phenomena as provisional objects, objects which can be described or apprehended in particular ways, but which escape definition or causal explanation. In some cases they are intentionally used to prevent the researcher from leaping ahead to complete explanations of what caused a phenomenon, a situation I will describe further in the next section.

This process of iterative engagement with an anomaly is like a process of reproduction in the sense of that term usually associated with the sciences. Reproduction, or more often replication, is often meant as the exact re-enactment of an experiment or analysis for the purpose of checking the results or demonstrating findings (justificatory purposes). However, work has shown that reproduction is also often about extending prior experiments, about doing something in a similar way but also differently (Feinberg et al., 2020). In this sense it is closer to a notion of reuse or repurposing. This is a rationale for extending or building off of prior work, but also for demonstrating the robustness or persistence of a phenomenon across slightly different contexts (Schmidt, 2009).

Whether we refer to this as reproduction, reuse, or repurposing, my argument is that we can consider it as an iterative, generative kind of improvisation (Weick, 1998). As Weick (1998) has pointed out about improvisation, improvising does not mean generating new actions randomly from whole cloth, but rather drawing on routinized activities and learned, structured protocols as resources in the assembly of new activities. In this same sense reproduction of an anomaly may

be an unplannable new engagement with a phenomenon, but it draws on established, routinized protocols and techniques. This highlights on the one hand that although Noah and the rest of the Radio Group are doing something new, they are drawing on a great deal of routinized activities and resources. The general approach of detecting, modeling, and flagging RFI is an established practice in the group. Plotting nascent phenomena against the dimensions of time, frequency, power, and directionality is a common technique in the group for giving these anomalies shape and character, and Noah reuses large swathes of the group's analysis code as well as Mason's previously developed software in order to perform these techniques.

At the same time the tests that Noah and others run remain open to the generation of unexpected things. The literature on routines and improvisation render these activities as performative, meaning that a routine or re-enactment is always a new performance and has the capacity for producing new things. In this sense although routines are understood as a kind of repetition, and can be a source of organizational inertia and inflexibility, they can also be the site of change and novelty (Feldman and Pentland, 2008).

This is where the notion of reproduction should place us, in a space of doing new things through the variation and extension of established tools. By varying and reconfiguring established resources and protocols in new ways, the Radio Group is able to produce new enactments of phenomena. They are able to transform anomalies and problems into new kinds of things. This is similar to what Rheinberger (1992a) means by a notion of differential reproduction. For the Radio Group, and for an increasing number of research groups across the sciences, this kind of differential reproduction is done in large part through the variation and recombination of software. Whether we consider software to be part of a repertoire (Ankeny and Leonelli, 2016) or grammar (Pentland and Rueter, 1994; Olson et al., 1994) for scientific work, we must consider it as a key part of the essential resources with which the Radio Group can assemble new enactments of physical phenomena.

It is important to outline this iterative process of reproduction because it is a particular kind of strategy of work that is geared towards working with epistemic objects. It involves sequential running of tests which cannot be made according to a plan, but which can be strategized based on past outcomes and which can take on meaning in relation to those past outcomes. This is clearly visible in Figure 5, where Noah has plotted the results of his algorithm (on the far left) next to the results of cosmo-rfi (second column from the left) on the same data. His new algorithm shows a clear detection of the feature, but juxtaposed with cosmo-rfi's results it is possible to see that cosmo-rfi saw the phenomenon similarly, but much more faintly and without detail. The character of a phenomenon is established not through one conclusive test but through its trajectory of development across many tests: its responsiveness to different kinds of probes, its record of differential appearance or non-appearance. As Weick (1998) indicates, citing Gioia's (1988) discussion of jazz performers, the improviser cannot look ahead and work from a plan, but can look backwards at notes that have just been played. In this case those 'notes' are prior enactments of phenomena accomplished through the variation of software components as well as many other potential aspects of the group's experimental system.

A closer look at the use of software in this reproduction process is warranted. In the next section, I will examine how exploratory work leverages the performativity of writing and running code.

4.2 Exploratory testing

The process of reproduction I described in the last section hinges on the ability of software tools to produce novelty. When code is run, the interaction it produces cannot simply meet the criteria of predefined requirements; it needs to have at least the potential to produce outcomes that are outside the present working order of the system. The research software system needs to function as a "generator of surprises" (Rheinberger, 1992a). How do researchers use software

in this way? The short version of the answer that I develop here is that they use software to perform exploratory tests. Members of the Radio Group in fact call individual iterations of their analysis process “tests.” Often what the Ph.D. student or research scientist is doing from week to week is running a test or a series of tests. These tests, however, different somewhat from most kinds of tests recommended as good software engineering practice. They do not have a gold standard against which the outcome of the test will be evaluated, but rather the outcome will be the point of departure for a sensemaking (Weick, 1995) process. In other words they run the test in order to *see what happens*. It is in this sense that the researchers’ tests are open ended (see also Kery and Myers, 2017). This can be contrasted with running software in a contractual way, where the proper operations and outcomes of the software have been defined, and its status as a working system can be evaluated against those gold standards.

The idea that exploratory tests produce novel events could suggest that running code in a contractual way cannot produce novelty. This is certainly not the case. The stance I take here is that running software is a performative act, meaning that the running of the software is always done again in slightly new conditions, with new data, new parameter settings, a new researcher in the chair, a new version of an underlying dependency, new hardware, or some other minor shift in context. For this reason even established software and well-tested software can break its contract, so to speak. It might diverge from its definition or requirements and result in surprising and novel outcomes. This is the nature of all kinds of bugs and breakdowns that occur in computing systems broadly. However, the researcher does not typically wait for random arrangements of a system to produce novel interactions that might (or might not) be interesting. Rather, they actively vary components of the system with the intention of producing outcomes that will give them new information about an anomaly. The exploratory test is carefully designed and reflects the intentionalities of the researchers who design it.

The content or shape of an exploratory test is always specific to the systems the researcher is working with, and does not lend itself to generalization. However, in this section I will walk through an example involving some of the Radio Group's techniques for running tests in open-ended ways. This project, which was pursued by a Ph.D. student named Stella, was also focused on detecting radio frequency interference, but it began unexpectedly following the breakdown of the cosmo-rfi software. As described in the last Chapter, the cosmo-rfi software was the product of Mason's prior work on RFI detection and had become an established tool for detecting RFI for the group. Mason had put the cosmo-rfi code into a repository on Github, and had applied some of the practices that the group associated with production software, including unit tests, some documentation, and a process for creating and reviewing code contributions ("pull requests"). The cosmo-rfi software had then been used by a number of other Ph.D. students in the group to identify and flag RFI in their data. It would typically flag some portion of the data as containing RFI, and the researcher could then remove that data and use the rest for their analysis.

The breakdown occurred when Stella used the cosmo-rfi software to flag a dataset that had been collected much more recently from one of the group's arrays. Stella's primary research topic was not on RFI at all, but rather on artifacts introduced in the low level signal processing stage. However, in order to evaluate the effect of her work on that topic she needed to bring together the group's whole analysis pipeline to perform a full analysis of the data. This involved running cosmo-rfi to detect and remove RFI, but when she ran the software on her newer dataset it did not perform like it had in many prior cases. It flagged almost all of her data as contaminated, leaving her nothing to analyze. This threw Stella's (and the group's) extensive instrument and processing pipelines into jeopardy: it could be that something had changed with the hardware of the instrument, or that some bug had been introduced into the code, or that the "RFI environment" at the telescope site had gotten worse due to the introduction of more radio-

emitting sources. The group knew that the RFI environment was getting worse in general, due for instance to the growth of Starlink satellite constellations (see also McDowell, 2020), but such a huge change seemed unlikely.

This discrepancy between past flagging operations and Stella's new effort was an anomaly. It contraverted the established working order of the cosmo-rfi software, and an explanation was not readily apparent given prior understandings of the RFI environment and the operations of the flagging software. Importantly, this anomaly appeared in software that was understood to be in working order. While RFI detection is far from a settled issue, the cosmo-rfi software was well tested and its general functionality was thought to be well understood. Stella initially ran the cosmo-rfi software in a contractual way. The software has an understood or expected outcome, and its status of working or not working can be evaluated against that expectation. If the outcome converges with these expectations or requirements then it is in working order, whereas if it diverges from them then it is in error. When confronted with a new dataset, however, this well established software produces unanticipated results that throw into question the issue of whether the software is working or not. There was then some uncertainty about the proper interaction that the flagging software should have with the RFI environment, and this throws into question both the software and the environment. This is what I mean by the fundamental performativity of running software, that even well stabilized software, run in a new context, can produce novel interactions.

Given the appearance of the flagging anomaly, Stella switched from a contractual orientation towards the cosmo-rfi software to an exploratory orientation. She wrote some plotting code which would plot her data against a number of dimensions that are salient in rendering interferometric phenomena: time, electromagnetic frequency, power (the strength of the reading on the antenna), and polarization (representing directionality of the reading) (Figure 7). Additionally, she plotted these next to plots showing what parts of the data cosmo-rfi had

flagged, as well as the category of interference that the software had identified. This allowed members of the group to see what kinds of things had appeared in the data, how cosmo-rfi had categorized them, and what specific parts of the data it had flagged. These are not accidentally-chosen dimensions to use in rendering the data. They are typical modes that the group uses for diagnosing or characterizing emergent phenomena. Understanding a phenomenon's shape in frequency and time, for instance, can go a long way towards characterizing what kind of thing it is. Something which moves in frequency is unlikely to be a TV broadcast station or a satellite, which usually emit in a specific range of frequencies and do not deviate. Knowing whether a feature varies by directionality might help distinguish whether it is a source of interference on site or a bug in the data processing.

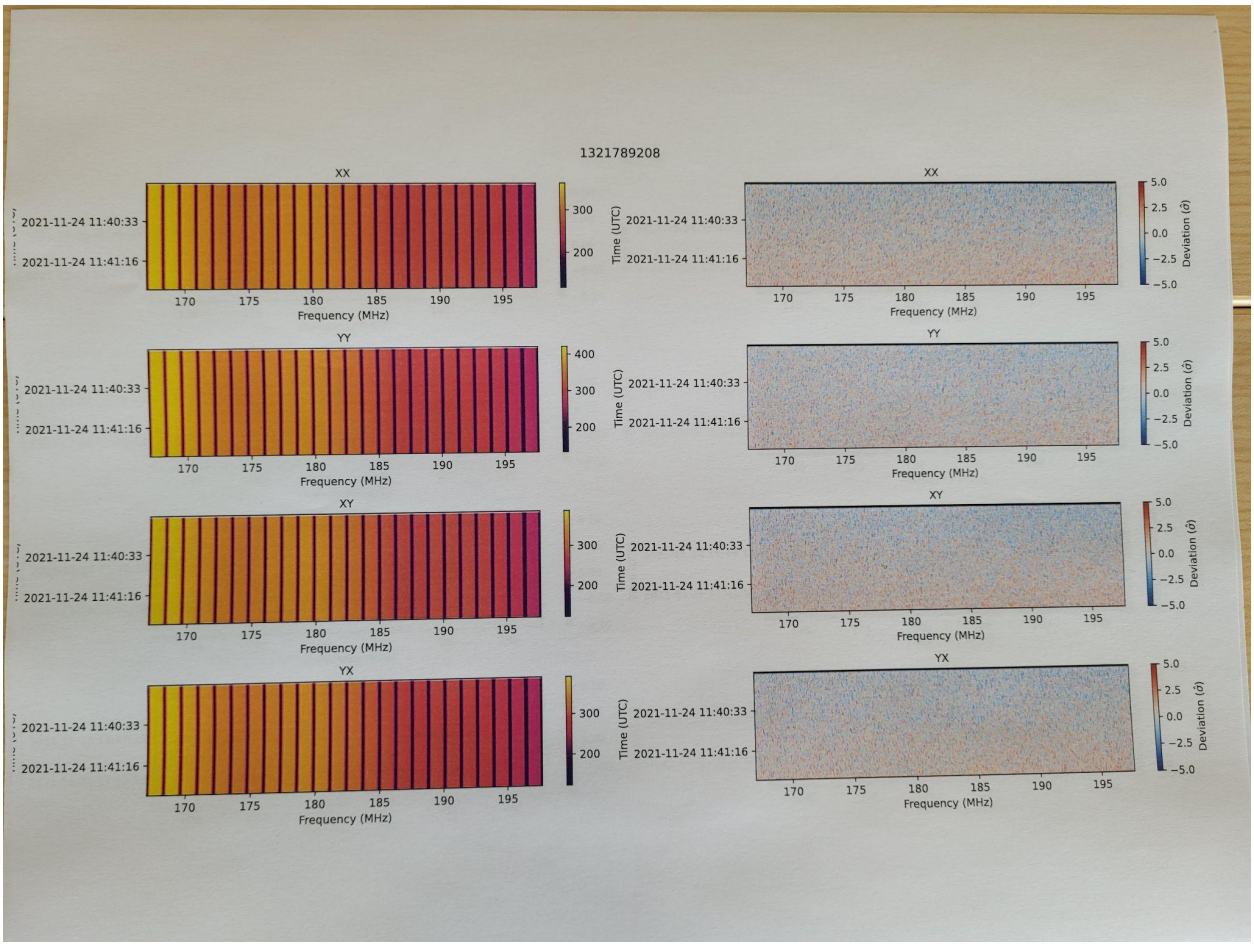

Figure 7: A collection of plots Stella produced in order to investigate the flagging problem. The four on the left show raw data, and the four on the right show the data after the application of the flagging analysis algorithm. The four distinct plots on each side show the same time sequence, but show different polarizations (reflecting directionality). If a phenomenon varied by direction it would appear differently in the vertically-oriented plots. The plots on the right display the “breathing” phenomenon, a gradual progression from blue to red.

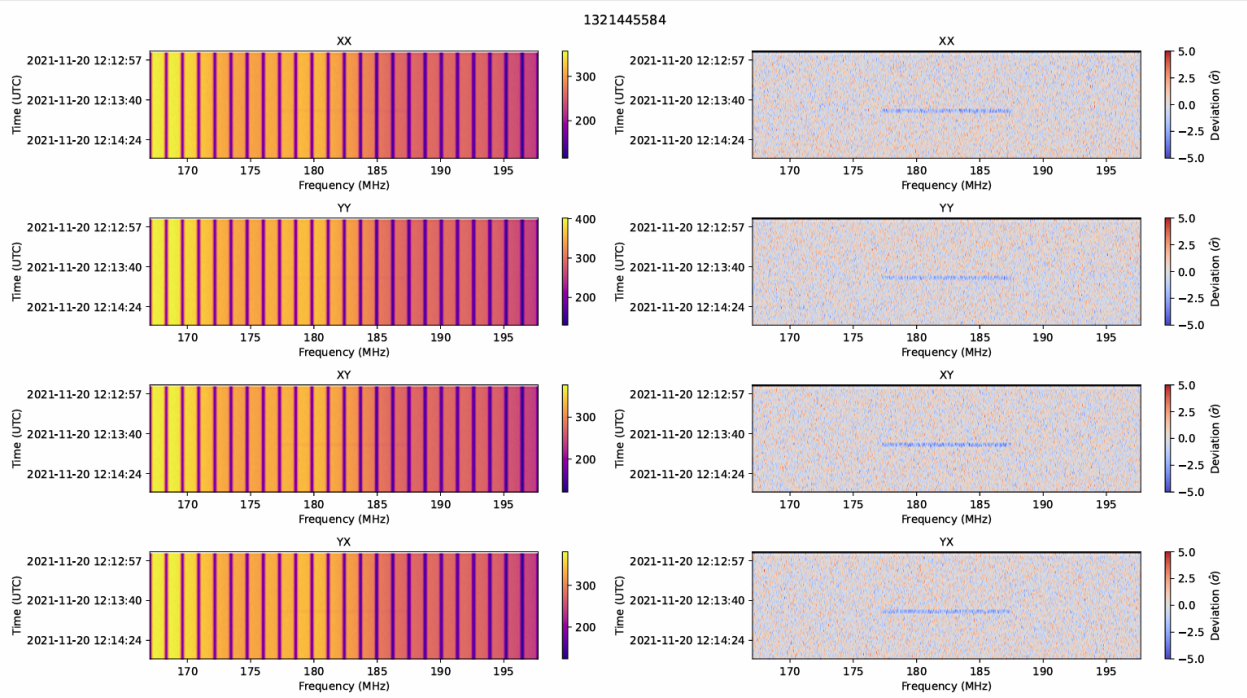

Figure 8: Plots similar to those in Figure 7, in this case showing the “one third line” phenomenon on the right hand side.

There were two significant features which appeared in relation to Stella’s test. One was “breathing”, which appeared as gradual changes in the power of the data over time (Figure 7). Another was the "one third line" (Figure 8), which was a blue line, indicating a strip of low power (Stella called it a "negative chunk"), which was very thin in time but extended horizontally in frequency. In every case this blue line extended across exactly one third of the frequency band. The one third line was a new phenomenon to them, and its consistent and sharp extent in frequency was strange. Breathing was in fact a feature that the Radio Group was well-familiar with: it was a pattern that appeared in the data due to the functioning of cooling units on the antenna array itself. Data from subsets of the antennae were collected together at intermediary

points across the array for preliminary processing, and these intermediary points had cooling units to maintain the temperature of the computers which were performing the processing. The breathing feature appeared in the data because these cooling units would turn on and off over time, changing the temperature in gradual ways, which affected the analog processing of the data. The presence of the cooling units therefore showed up in the data as gradual rise and then fall in power over time, across all frequencies (Figure 7). While this was a well understood phenomenon, the group noticed that where there was breathing the flagging software was throwing out the entire observation. Breathing was a relatively benign feature which did not warrant tossing out all of the data affected. The anomaly in this case was not breathing itself but an unclear interaction that it was having with the flagging software.

In what way was this second test of Stella's exploratory? As I stated above, the specifics of what makes a test exploratory are always entangled with the specific materiality of the instrument being leveraged, and they will vary depending on whether one is using a centrifuge, a magnetic needle, or a piece of flagging software. Stella performed a kind of *battery testing*, in which the interaction between the flagging software and the data was rendered against a variety of different parameters that are typically significant in characterizing interferometric phenomena. This is specifically a kind of test to be done with a poorly-understood anomaly. The group did not have a specific hypothesis to test, and as I will discuss below they actively avoid such hypotheses in these early stages of analysis. Their goal is more open ended. They can render the data against the dimensions of frequency, time, power, and directionality in order to give shape to an anomaly that does not yet have much shape at all. In this sense battery testing is specifically tuned to open-ended investigation. The breathing and one third line anomalies are new transformations of the original flagging anomaly that have emerged in response to a broadly (but strategically) constructed probe.

The difference between running software in an exploratory or contractual way is a matter of orientation, and we can see the difference in the way that the researchers operate around the test and its outcomes. The outcomes of Stella's tests were analyzed through an activity that the group calls a "data rampage". The data rampage involves laying out a very large number of plots across multiple tables (Figure 9), often in columns by night, and within columns by time (Figure 10). This setup allows the group to walk up and down along the table as a group and look across a large amount of data at once, looking for patterns and categories of phenomena. Combined with the dimensions rendered on the plots themselves (time, frequency, power, directionality), this arrangement scaffolds the use of the output of the software as a site for sensemaking. Group members can point each other's attention to particular examples, and once a category has begun to be described members can hunt across the other plots quickly, looking for other instances.

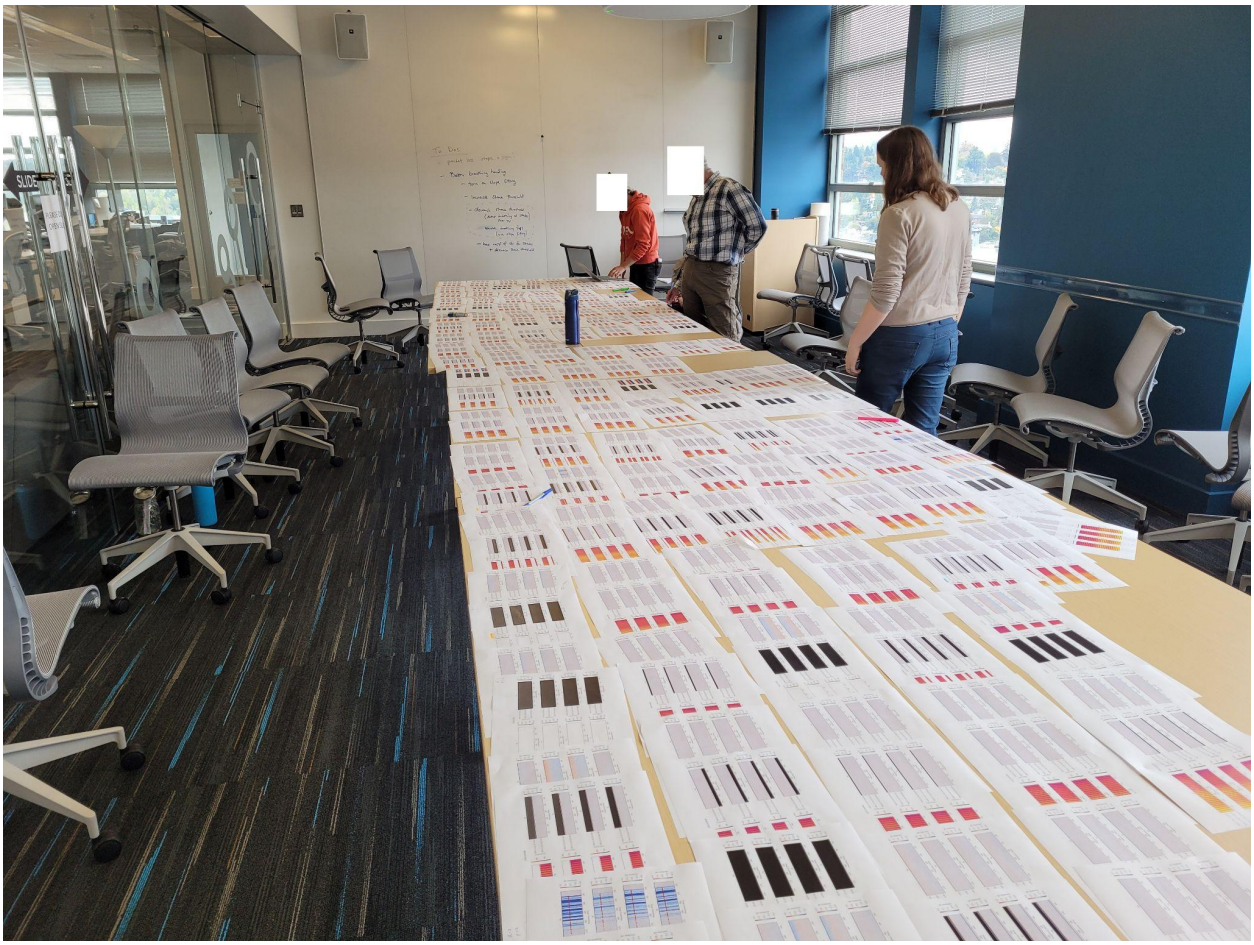

Figure 9: Photo of a data rampage. Plots are laid out on the table in columns by observing night, and proceed vertically in time, such that plots at the bottom of the table (the right side in this photograph) are later in the night. Group members have written “to dos” on the whiteboard at the back.

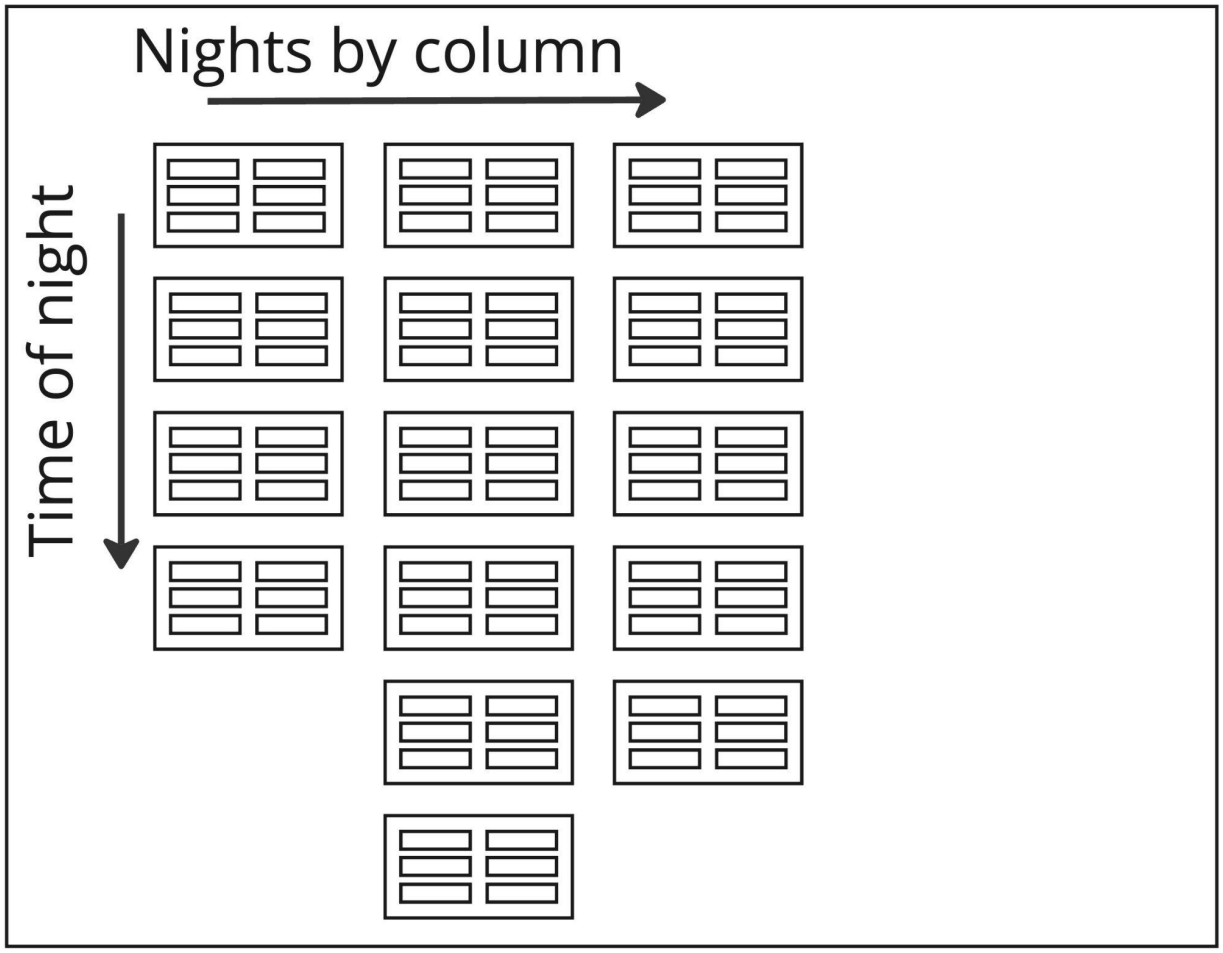

Figure 10: Schematic diagram of the layout of plots. Each column contains data from a particular night, and within each column plots are organized by time.

The data rampage proceeded first with people looking for *kinds* of features, features that had similar characteristics and could be grouped together, which would result in categories that would be written up on the whiteboard. In Stella's case these included the one third line and the breathing phenomenon. The Radio Group regulated their interactions around these novel phenomena in particular ways. At the outset of one data rampage, Magnus explained the process for new students in the lab and for my sake. He said that the idea was to "let the data speak to you", and that they would not be talking about "theories." By theories he did not mean grand physical theories, but rather explanations about what interactions or phenomena had caused the features they were seeing in the data. The rule against developing explanations or

"theories" was to slow down the process of interpretation. During one data rampage I transgressed the moratorium when we looked at some patterns of "breathing" in the data. I had seen that feature in an earlier data rampage, and so when they referred to it as "breathing" in this instance, I quickly suggested "isn't that the refrigerators?" Magnus and Mila both nodded, but Magnus said that was "usually" the case, adding back a qualification, and he and others continued to use the term "breathing." I realized that I had provided a theory rather than remaining on the level of phenomena-in-plots, on the level of "breathing."

After categories had been largely developed in the data rampage the group would discuss "things learned" about the features, and planned next steps. "Things learned" included, for instance, that certain features appeared in certain frequencies or that they often appeared over long periods rather than brief occurrences. Members of the group would discuss this and then plan new tests. This involved planning tweaks to an algorithm that the student had written or using different parameters with the group's primary analysis software. It also might involve selecting particular nights of data where there were a lot of features for further analysis. These practices were present in different ways in the group’s weekly lab meetings, which would often consist of group members standing around a Ph.D. student's computer, examining plots they had produced, and discussing what they might try next.

Further investigations allow for further transformations of the problem. For the "one third line" this was fairly quick, as Magnus, the head of the lab exclaimed during a meeting that it was likely "packet loss." This was a likely explanation because there was a part of the processing chain on site in which data was separated into three cables by frequency, and so occasional dropped packets on one of those cable connections would neatly explain the extent of the line in frequency, its brevity in time, and its low power. Similarly, through further tests after the data rampage Stella was able to establish that the breathing feature was being miscategorized by cosmo-rfi as a television broadcast, which always required throwing out the entire observation

that had been contaminated. The sudden appearance of this issue in Stella's data was attributed to the replacement of an essential processing component on the array. The new model was theorized to be producing new kinds of breathing patterns which had interacted with the cosmo-rfi software's established thresholds for identifying TV broadcast features. This meant that a relatively benign feature was being miscategorized as a pernicious one and a great deal of data was being flagged unnecessarily. These observations again transformed the Radio Group's flagging problem. The "one third line" became "packet loss", and the "breathing" interaction became a specific kind of miscategorization event.

It is with these framings that Stella's work with the flagging software returned from exploratory to contractual. Once the "one third line" has been transformed into "packet loss", an available course of action is to simply flag this new kind of feature in the data. Stella made a pull request (she contributed some code) to the cosmo-rfi repository, in which she was able to express her requirements in fairly specific terms: "Provide the option for flagging shapes only if the computed significance value is negative." This would enable cosmo-rfi to check whether a feature it detected in the data was negative, rather than just considering an absolute value, allowing it to detect instances of packet loss specifically. Not only were the requirements for this piece of code now clear, but Stella also now oriented towards the output of the code in a contractual way: by simulating a packet loss feature she is able to write a unit test that defines whether the code is working. Mason reviewed the code and it became part of the cosmo-rfi software. In a similar way, once the breathing feature was understood in terms of miscategorization, Stella was able to change the thresholds that cosmo-rfi used to detect different categories of interference. These changes collectively allowed her to use around 75% of her data, resolving her initial problem of having no usable data.

With Stella's contribution back to the cosmo-rfi software, she closed one full arc of a research project on RFI: the appearance of a surprising anomaly, the generation and refinement of 'the

problem', and the cosmo-rfi software's return to working order. There are a couple of aspects of exploratory testing that become clear in this project. First, running even established or productionized software can produce novelty or surprising anomalies. Software can be stabilized and standardized to a great degree, but running software is a fundamentally performative act. Just as with other kinds of artifacts, the software is not just a container for already conceived mental models (Schmidt and Wagner, 2002). The running of software has the potentiality of producing surprising outcomes. That said, the notion of the exploratory test turns on the idea that the researcher can actively prompt or seek out emergent anomalies that escape their present understanding. Anomalies can emerge not as unsolicited surprises but rather in response to tests designed to elicit them. Although the outcome of the exploratory test cannot be predefined, they can be actively designed around epistemic objects, things which are characteristically incomplete and poorly understood. The researcher therefore orients towards that output as something to be learned from, the point of departure for making sense of a larger situation.

Despite this open-ended orientation towards running the software, Stella's example demonstrates how these novel and poorly understood outcomes are nevertheless (and necessarily) produced using established routines and techniques. The one third line and the strange interaction between breathing and cosmo-rfi became visible within plots constructed according to established techniques within the group. This includes a great deal of underlying software dependencies, such as the plotting libraries that correctly align axes on plots, for instance. However, as Ankeny and Leonelli (2016) point out, researchers' "repertoires" include not only material resources but also the experience and know-how in bringing them together. In the Radio Group the ability to develop and design plots is a conspicuous skill necessary to constitute and reconstitute phenomena (see also Paine and Lee, 2017). Moreover, the practice of plotting data according to particular dimensions, as well as using different algorithms for

rendering features in the data are all techniques that, although established or routinized in the group's work, can be used to produce new phenomena.

Considering these aspects of routinization or technique, it is clear that exploratory testing is not a matter of random flailing about. Exploratory tests are carefully and intentionally designed, and it is as much this intentionality and systematicity as their novelty that makes them useful. Without continuity with past ways of working and established techniques any novelty produced in running the code is meaningless or incoherent. The one third line or the evil cow, for example, are novel phenomena but they bear continuity with established modes of engagement, plotting, rendering, and representing in the Radio Group's longer trajectory of work.

Another aspect of exploratory testing is that it entails retrospection. The general goal is to run the software and see what happens, which means instrumenting some particular interaction or event such that what happened can be apprehended in some way. This is the reason that the Radio Group has such an emphasis on creating and inspecting plots, because it enables them to reflect on what has occurred in past action. This retrospection is an essential part of sensemaking and other pragmatic understandings of design and organizing (Weick, 1995; Schön, 1983). Here it is also a practice and concrete aspect of what an exploratory orientation towards software entails. In general the process of exploration is aided by any scaffolding that can be assembled around the interpretation of past action.

There are also a couple aspects of the larger exploratory process which are exemplified in Stella's work. Firstly, it highlights a distinction between exploratory and contractual orientations towards software, and the way in which those orientations shift over time. The cosmo-rfi software began as code written in Mason's exploratory work on RFI, but as he reached greater confidence in what it was able to detect and remove from the data, he turned the software into a more robust tool that could be reapplied or repurposed again and again by other members of

the research group. In this way the software becomes a technological object, as Rheinberger (1992a) puts it, which can be reapplied in new exploratory settings. With Stella's initial flagging discrepancy, the cosmo-rfi software once again came under issue. Given the erratic outcome of the flagging operation the group can no longer take it as assumed that the cosmo-rfi software is in working order, at least not in the context of Stella's new data. They then had to reorient towards the software, treating its outcomes as the things to be figured out. With Stella's eventual contribution to the cosmo-rfi repository, the software came back, once again, into working order. A point of significance here is that code does not move unilaterally from problematic object to production tool, it is reproblematized and re-productionized in an ongoing way. This is a kind of vacillation between the consideration of phenomena as outcomes of the group's instrumentation and the infrastructural inversion (Bowker, 1994) of that infrastructure, which involves re-examining parts of the software and taking its functioning as problematic.

In doing exploratory testing the researchers are working with software as an ensemble (Howison and Herbsleb, 2010). Stella's initial goal was to test one piece of code that she had written in the course of her work on low level artifacts in the signal processing chain. However, 'the software' is always many things operating together, and in testing one part of the software another part broke down and became problematic. In order to examine or isolate one part of a much larger codebase for inspection, the researcher needs to be able to rely on many other components working as expected. Every test the group runs relies on a vast network of operations, such as basic mathematical transformations performed by underlying mathematics packages, and the correct rendering of data along axes performed in plotting packages. This is a relationship between stability and flexibility in infrastructure that I will return to in the next chapter. Here it is enough to point out that exploratory work involves problematizing and reproblematizing different components of a larger infrastructure over time.

In returning to working order, cosmo-rfi is not returning to the same working order that was obstructed by Stella's initial discrepancy in flagging. The cosmo-rfi software is now newly capable, or rather the capacities of action that the Radio Group can undertake with their collected instrumental arrangement has changed. The cosmo-rfi software can now identify and remove new kinds of interference. This will ultimately, they hope, allow them to see something else, the 21cm signal. The instrumental system has become more refined in the sense that it has become more discerning in its engagement with celestial objects. The interaction between the instrument (interferometer and software) and electromagnetic radiation has come to reflect a more refined intentionality on the part of the researchers about what it is that they want to capture.

4.3 Iteration and the indefinite rhythm

Despite its inherent unpredictability, research work can be structured temporally. By "structured" I am referring to specific patterns of routinized activity, by which the Radio Group organized their work in a way that is responsive to the unpredictabilities of their research. In particular members of the group put a premium on rapid iteration and they carried out a kind of *indefinite rhythm*, in which projects were not timebounded but proceeded in an extremely regular cadence. I argue that these are patterns of activity which are intentionally suited to facilitate the retrospective processes I described in the last section, and to work through the problem of uncertainty.

Because the group's work is built on the production of surprises, they must deal with a great deal of uncertainty in the direction and length of their projects. Danielle, another student in the group, described this dynamic:

> "Every time something surprises me, I need to do something new about it. Every single time my data looks different than I'm expecting. I need to ask a question of it I wasn't expecting to ask, which requires new code" (Danielle, Ph.D. Student)

Danielle here captures the dynamic of needing to redirect work upon seeing the outcome of an analysis. She also related that her own primary research project had been intended to be a one year effort towards the beginning of her Ph.D. career, but it had turned into a project that spanned her Ph.D. career:

> "But all of that was always with the purpose of making the [pipeline] that our group uses work for [dataset] and be able to do an imaging power spectrum. That was always the goal of what I was doing. So it's been kind of an iterative process of let me run [the dataset] through [the pipeline], see how things are looking, and then every time I would do that, I would find some sort of problem with the data and it would kind of circle me back to the commissioning side of things. And so then I would kind of iterate, be like, oh, let's go figure out what's wrong with the telescope that's causing this. Try to get the commissioning team to work on it; Lather, rinse, repeat, try again. And so the initial goal, I think when I joined was that I would create an imaging power spectrum in the first year, and I'm yet to make one because there were so many problems..." (Danielle, Ph.D. Student).

Here Danielle describes running data from one antenna array through the Radio Group's primary analysis pipeline, which had been developed for data produced by another array. They had expected this to take about a year, but because of all of the unexpected problems they had encountered it had taken much longer. The group is often engaging poorly-defined problems, and their work proceeds on the basis of interpreting outcomes of exploratory analysis efforts. For these reasons it is difficult for them to predict or plan the direction or timeframe of their projects. In other words, because the group’s exploratory work proceeds through the production and interpretation of surprises, their work takes on a process-level uncertainty in direction and timeframe.

This situation represents a particular kind of unpredictability that plays out across successive testing activities. As in Kery and Myers’ (2017) description, this work involves a great deal of backtracking or changing direction, but it is also important to point out its seriality. The group's

work becomes serial in the sense that one analysis has to follow a previous analysis. Often this was done through a rhythm of weekly lab meetings, in which students would bring results from the past week's analysis efforts to the lab meeting, the group would collectively evaluate it, and based on that evaluation they would plan the next week's efforts. Additionally, this seriality was necessary because members of the group did not know what later steps would be until they interpreted the outcomes of the present step. This is different from seriality produced by a kind of functional dependency, where, for instance, one task might rely materially on the output of a prior step, but the different steps and their order are relatively well-defined. The Radio Group, however, drives their work through the engagement of surprises, meaning that future courses of action only take shape in the retroactive evaluation of current testing activities. As Danielle put it above, they need to ask a question of the data they were not expecting to ask. It is in this sense that the seriality of the work is epistemic.

It should be noted that this seriality is a matter of degree. There were often situations in my observations in which researchers could pursue multiple lines in inquiry at once, especially if they were investigating multiple different problems. Additionally, members of the group, especially more senior members, often had hunches or vague plans about what they would try next. Nevertheless, furthering a given line of inquiry always required successive iterations of running code and interpreting outputs, and the redirection of projects based on these outputs was a regular, and often expected, event. Indeed, uncertainty was a matter of course for members of the lab. During one lab meeting a student reported that he was working on a particular mathematical transformation and said, cautiously, that he thought this was probably the last problem and then he could move on. Magnus joked that that is "what keeps experimentalists alive, the idea that it's always the last problem." This joke plays on precisely the seriality of their work, the idea that future problems are obfuscated behind the current one.

Because future action was dependent on the outcomes of present tests, there was an emphasis on how quickly they could try something new and *see what happens*. Magnus described this in comparing the work of the Radio Group to another research group he was familiar with:

> "...so we don't- we have a much tighter computational budget, but we are optimized for testing speed. And they have just been hampered by the fact that, you know, when we're in major cycles we'll be testing two or three changes a day. For them a test is a two-week process. It's just a lot harder" (Magnus, Professor).

Here Magnus is referencing the fact that the Radio Group works in higher level programming languages (IDL and Python), whereas the other group's work involves a much lower level programming language. They also have a much larger budget for running large scale computations. Despite this benefit, as Magnus argues, the other group has a hard time iterating on an analysis and running large processing jobs to get back results. For him this is the main criterion: the ability to iterate on an analysis, get back results, and learn from them. Moreover, the importance of this process of iteration is precisely connected to the uncertainty of the work:

> "If we had known what we were doing from the beginning and had specs and just wrote it down they would have won, because they have a higher compute budget [...] but because we don't know what we are doing [laughing], testing becomes the limiting step" (Magnus, Professor).

Because the group's work has a high degree of unpredictability, both at the individual analysis level and at the larger process level, the ability to try new things quickly and see what happens is a primary motivation. As Magnus says, the group's work is "optimized" towards this purpose. This is a similar focus on rapidity of iteration embedded in the notion of "knowledge turns" discussed by Goble et al. (2011).

This process of iteration typically unfolded through a rhythm of weekly lab meetings. Lab meetings were not unlike the data rampage described in the previous section in that they were an occasion for the group as a whole to try to make sense of the students' work over the last week. Meetings typically proceeded with the students taking turns to show plots, often on their

laptops with others standing around looking over their shoulder. Senior members of the group in particular would try to interpret the outcomes and then suggest new tests to run for the following week. This constituted a rhythm of testing, retrospection, and the projection of new tests through which the process of reproduction was organized in time. In some cases the iteration needed had to do with the plotting of the data or something else that could be changed rapidly. In these cases students would sometimes rewrite the code on the spot, while the lab meeting proceeded to other students, and they would return to that student at the end. From my perspective this always seemed like a lot of pressure, to try to rewrite complicated plotting code in real time, but being able to see and interpret the results meant that they could figure out what to do next, and so the quicker that they could see new results the more quickly they could figure out next steps.

While this process of iteration had a great deal of uncertainty in direction and timeframe, the pattern of lab meetings made it extremely regular in time. Students did not always have new plots to show, and on a few occasions lab meetings were canceled, but for the most part the lab meeting provided a regular structure to research work across my 6 years of observations. In this sense it was an *indefinite rhythm,* which was not timeboxed to a definite deadline, but which was regularized in cadence.

4.4 Some commitments of an examination of exploratory programming

Across the previous sections in this chapter I examine the Radio Group's work as a kind of exploratory programming, and I outline some of its processes, goals, and rhythms. A large part of my goal in this dissertation is simply to make exploratory programming visible as an intentional kind of work that researchers do, which has some of its own processes and criteria distinct from software production. Here I will summarize some of the commitments of my

account of exploratory programming in the research process, describe what I mean by making it “visible”, and weigh some of the potential implications of that visibility.

The first commitment of this account of exploratory work is the idea that anomalies and problems are points of growth or change for a system. Although the researcher may express exasperation or frustration with a particular problem or anomaly, they operate around them as their primary resources for organizing research projects and designing new tests. In a practical way, problems and anomalies serve as starting points for new projects, they are the basis for designing new tests, and they serve as handles or provisional caricatures of phenomena-in-the-making. This is an ‘anomaly-positive’ model of scientific work in the sense that it positions anomalies not as hurdles that must be overcome but rather has emergent handles that researchers use to grasp and manipulate as of yet unknown objects. I take this stance in part because I am looking at somewhat prosaic kinds of anomalies. Kuhn (1962) acknowledged the ubiquity of anomalies in science but focused to a large extent on the persistent, paradigm-breaking anomalies that cause widespread divergences and conflict in scientific communities. This is a paradoxical role of anomalies that I do not attempt to untangle here. Anomalies are unwelcome to scientists in the way that they overturn and trouble existing understandings, but in precisely that role they are also resources for doing new science and for securing scientific prestige. Nothing is more important in science than having a significant and meaningful problem to work on.

In my account I have used the terms production and reproduction, which are an odd way to describe the emergence of anomalies and problems. Rheinberger (1992a) also confesses that he is somewhat uncomfortable with the word, but uses it nonetheless. I also use it because it captures the intentionality with which researchers bring anomalies into being, despite the fact that they do not know what they will look like. Moreover, it captures the fact that anomalies and phenomena are enacted and not just stated or theorized. Researchers use a variety of tools to

make phenomena, anomalous or not. This is not to assign complete agency to the human in the situation, but rather to emphasize that the phenomenon produced is an outcome rather than a readymade resource for scientific activity. As in Kohler's (1991) usage, these aspects of the word production do come in some way from its reference to systems of producing material goods, but it should not imply any particular economic theorization of scientific work. Its commitment is more towards science as practice (Pickering, 1992), and the day-to-day manipulations of instruments as epistemically potent aspects of scientific work.

On this last point concerning science as practice, my account of exploratory programming takes software in particular terms. I am taking software in terms of its workability (Spencer, 2015), or as a manipulable system (Turnbull and Stokes, 1990). This involves looking at how researchers turn software into a set of tools which can, in a better or worse way, facilitate the ongoing manipulation and refinement of their objects of investigation. This framing finds overlap with the concerns of software engineering more than with many other areas of computer science. It concerns in particular the way that people work with software as a material for making things, how they manage its multiplicity and its interconnectedness. This is not, for instance, an examination of a digital transformation of scientific work or an analysis of the formal representation of scientific concepts in code.

As with other treatments of science as practice, this account also goes some way towards demystifying the processes of science. Like Star and Gerson's (1987) account, my account of researcher's engagement with anomalies casts it as a relatively prosaic thing. Engaging anomalies through software looks a lot like simple debugging or troubleshooting. I think this is a feature rather than a bug of the consideration of science as practice, so to speak. Debugging and troubleshooting are terms that have come to be largely reserved for interactions with software, but doing science with air pumps (Shapin and Schaffer, 2011), for instance, involves a great many mundane activities that could be described as troubleshooting or debugging. The

apparent strangeness of considering debugging as scientific activity has more to do with the longer-term obfuscation of technician work in the literature (Shapin, 1989; Barley and Bechky, 1994) than with some fundamental division between IT work and science.

Outlining these characteristics of exploratory programming help make it visible as an aspect of scientific work. By “visible” I mean both that we can take exploratory work seriously as its own process with distinct kinds of criteria, but also that we can make software visible as an epistemically potent tool in scientific work. On the first point, examining exploratory programming work in the sciences is important because when scientific software is discussed in the literature it is often discussed in terms of software production. This is understandable given the urgency of problems around sustainability, reproducibility, and reuse that processes of software production often address. However, the contingencies and unpredictabilities of exploratory work are often couched as obstacles to the central effort of software production. One of the benefits of focusing on exploratory work itself is that, for instance, it can be informed or designed for. The improvisational and contingent processes of exploration can take on some of the qualities traditionally associated with the context of discovery in the sociology of science, that it is a magical or arbitrary process intractable to rational analysis (Popper, 1961). Kery et al. (2017) have already made contributions to informing and designing for exploratory activity, and the organizing routines and techniques such as those used by the Radio Group are things that can be shared and adapted between research groups.

Making exploratory work with software can help us understand some of the mysteries that surround software production in the sciences. For instance, it is important to observe that the unplannability of scientific work does not inhibit them from structuring and routinizing that work. Indeed, members of the Radio Group organize their activities specifically to deal with high degrees of uncertainty in the running of tests. Greater engagement with the literature on organizational learning, which has described various configurations of a tension between

routines and improvisational work (e.g. Gersick, 1994; Obstfeld, 2020), is needed to further this understanding of research software development.

An important follow-on to this observation is that although the practices of writing code that researchers use in the research process are unorderly within a regime of software production, they are nevertheless orderly with regards to a process of research. Seeing the Radio Group's work with software as a structured research activity rather than an unstructured software development activity changes our view of what the path towards research software infrastructure might look like. Any intervention into the work of scientists that has the goal of improving the way they develop software will need to be reconciled with their established patterns and rhythms of work, which are intentionally structured and not orthogonal to the work of writing code. In the case of the Radio Group, programming is already deeply embedded in the research process. The uptake of new practices for software development, then, looks more like an integration of different kinds of practices rather than the adoption of development procedure into an unorderly or amethodical space. The details of this integration in the case of the Radio Group will be discussed in Chapter 7.

Without looking at exploratory programming, discussions of research software seem to start where the work of figuring out or investigating ends. This places a core aspect of scientific work outside of or away from software artifacts. My observations with the Radio Group stand in direct contrast to this view. Software is the means by which the Radio Group enacts the "working objects" (Daston and Galison, 2007; cited in Strasser, 2019) of their research. Even anomalies and errors must be made or produced within a research system (Star and Gerson, 1987), and they appear only in relation to that established system. More specifically, exploratory work is a process that makes novel or problematic objects "organizationally real" (Østerlie and Monteiro, 2020), meaning that they can be incorporated into organized processes and manipulations of a group. Seeing software in this way is an alternative to seeing it as a substrate for science.

Seeing it as a substrate means seeing it as a technical underpinning which is necessary, and could potentially slow or expedite science, but which is essentially discrete from the epistemic work of science. In contrast to this view, the writing and rewriting of software should be seen as constitutive of successive enactments of a phenomenon, and of successive framings of 'the problem.'

Last, and most important for my purposes here, making exploratory work visible allows us to ask questions about how this exploratory work fits in with, or becomes complementary to, processes of software production. For the sciences, whose work centers primarily on exploratory work but who are increasingly enjoined to take up software engineering practices, this is an urgent concern. I turn to this interrelationship in the next chapter.

Chapter 5:
Software production and the research software system

In the last chapter I argued that a process of exploration can not just produce new phenomena; it must do so within frameworks, routines, and techniques that establish continuity and coherence with a longer-running trajectory of research work and instrumental development. From this perspective we can reproach the problem of building research software infrastructure by asking how researchers both accomplish stability in their routines and software resources while also accomplishing flexibility in the pursuit of new research problems, new phenomena and new stakeholders.

In this Chapter I argue that the Radio Group does this by accomplishing a distinction in their work between processes of exploration and processes of software production. These are distinct regimes of work that operate on different temporalities, have different criteria for what constitutes good work, and pursue different goals. Where exploratory work proceeds through a kind of indefinite rhythm, as described in the last section, software production typically involves longer-range planning and may have time-bounded development goals. Where exploratory work prioritizes rapid iteration and the ability to make changes quickly, software production prioritizes the robustness of software across stakeholders, its sustainability, and its ease of use and interpretability. Where exploratory work is focused on changing understandings of phenomena and the capacities of the researchers' instrumentation, software production is focused on entrenching understood capacities in robust and sustainable ways.

These distinct regimes of work are mutually interdependent. What I mean by interdependent is not necessarily a strict functional interdependence, but that they are defined in part in relation to

the other and each benefits and is promoted by the other. For instance, software production does not only contribute to a software tool's sustainability or usability as a standalone product; it also contributes to and facilitates the process of exploration. Moreover, exploratory work feeds novel capacities into the process of software production, defining requirements for new development of established software tools. This latter dynamic I will take up in Chapter 6.

The distinction between exploration and software production is not a readymade, inherent distinction in kinds of work. Rather it is the outcome of boundary work (Gieryn, 1983). Members of the Radio Group and the larger CDA negotiate the boundaries of these regimes in an ongoing way, defining the when and the where of exploratory programming and software production. In particular, they refer to the categories of "research code" and "production code" (or "collaboration code"), objects of work which align with exploratory work and software production respectively. Over the course of the last decade the Radio Group and the CDA developed distinct associations and ways of working with these categories of software artifacts. I take these perspectives and actions as a point of departure for my own analysis of these categories.

Looking closely at this distinction between exploration and production also begins to tell us how the CDA came to integrate software engineering practices into their work. Rather than universally applying such practices wherever they work with code, they designate specific contexts to the use of engineering practices. This is the basis for talking about research software *systems* that contain within them heterogeneous and interdependent kinds of work. Specifically they encompass the mutually interacting processes of exploratory programming and software production. The notion of the research software system draws on Rheinberger's (1992a) notion of the experimental system in focusing on the generative potential of the interaction between well-understood, ready-to-hand infrastructural resources and fundamentally unfinished and problematic objects. However, the research software system is focused more on processes of work by which these interactions are enacted and re-enacted over time.

Such a system still entails significant work in the learning and adaptation of software engineering techniques and the extra work (Trainer et al., 2015) they require, a topic I engage more closely in Chapter 7. It also, however, entails articulation work in successfully connecting these distinct regimes. Articulation work (Strauss, 1988) is the work of connecting different tasks and sequences of tasks and ensuring that these different activities function well together. Making a research software system go well entails not just defining different regimes but also work to connect and transition between them. As I will argue, maintaining two distinct modes of work with software within a larger collaborative milieu brings a certain amount of complexity.

5.1 Production code, flexibility, and the wild west

Members of the Radio Group and the larger CDA generally made distinctions between "research code" and "production code" or "collaboration code." My analysis of exploration and production center around these categories respectively. At the center of these distinctions were rationales about the flexibility and rigidity of the work that they performed with software. These rationales were multifaceted and reflected a number of different stances on the tradeoffs of using software engineering processes. Here I will explore the distinction that researchers developed around the notions of research code and production code. In particular, researchers understood these designations and their tradeoffs in terms of the flexibility needed for highly uncertain, exploratory work and the rigor and consistency provided by software production methods. These were complex, multifaceted ideas, but they provide the basic rationales for the organization of a research software system, which attempts to maintain spaces for both production work and exploratory programming.

In examining the designation of production code, I will focus on a particular software package, called pycosmo, which became an exemplar of production code. Pycosmo was a Python package initially developed to act as a format interchange or "clearing house" for data formats,

as one researcher put it. Researchers in the field commonly used a number of different data formats to store and share data, depending on their own preferences or projects that they had previously worked on. Pycosmo was written to convert data between these different formats, accounting for different conventions in representing, for instance, the numbers of antennas or the polarization (directionality) of the antenna elements.

Pycosmo is illustrative of the broader phenomenon of production code for a couple reasons. Firstly, production code was closely associated with the scale of its use across a wider collaborative milieu. The interchangeability of the terms production code and collaboration code[2] is indicative of this. The idea of applying software engineering techniques to software work was closely dependent on the number of people who were expected to use it in the future.

While this association is not novel to wider discussions about research software, it is important to substantiate how these collaborative needs became a motivation for researchers in this case. The motivation for pycosmo had emerged at the beginning of the CDA, and it was associated with new collaborative demands that project created in the field. As Mila, an early developer on the project, put it, she and a number of other members of that project realized that conversion between formats was going to be a significant barrier, and that the existing scripts used for that purpose were “fiddly and not robust, research-level, one-off, get-the-job-done kind of scripts.” The format conversion process would need to be done consistently and reliably across the many researchers invested in this new project. Pycosmo was built with this purpose of data exchange in mind, but once it was built the developers realized that it was a good place to develop many other kinds of functionalities that one might want to use while manipulating, inspecting, and plotting data. As a generally useful tool, pycosmo came to be an essential

[2] “Collaboration code” sometimes referred to code specifically associated with one particular interferometer project, but it was also used to describe code intended to support wider collaboration more generally.

dependency for a great deal of the software in the CDA collaboration, as well as at least one other research group working in the field. It also was picked up by the High Altitude Low Frequency Observatory (HALO), another interferometer project that was not focused on 21cm research.

Spaces for the development of production code took on this quality, both of being intended for a large group of people but also being developed by multiple people. Pycosmo had around 5 permanent developers, both research scientists and professors, who all came from different universities. Many of them worked on the CDA collaboration but a number of them worked on the WRT as well and one worked on HALO. When I asked members of the Radio Group why pycosmo had such a great deal of software engineering practices used in its development almost everyone gave the rationale that it was intended to be used by a broader group of researchers.

Production code involved other people in the development process, but also required developing the code *for* other people. Mason, a Ph.D. student, described these considerations that came to bear on his own contribution to pycosmo:

> “And so you know, pycosmo has to take some special care to make sure that their software is compatible with what the people who depend on them are trying to do, which is really challenging, I think. Because… I needed this thing for cosmo-rfi, and it wasn't that hard of a thing. But what it means in pycosmo is I've changed this function so that it returns this output, which ends up adding these if statements everywhere. If statements are gross. And then not only that, but I have to go and take this object, and I have to give it a new attribute that's optional. And I have to make sure that thing is compatible with what everyone else does. It was kind of like everyone comes to town and they're just like, ‘Okay, how do we satisfy everyone's needs?’” (Mason, Ph.D. Student)

Discussions of potential contributions (pull requests) often included the meticulous details of how to phrase warning messages, what parameters people might want on a particular function, and how to order them in the most intuitive way. Contributions to the software had to include unit tests, which compared the output of the code against expected output to establish that it

was working as expected. These tests were run regularly as part of a continuous integration system, to check that any new code that was being considered for addition to the repository is not breaking expected functionality.

It was around this expectation of the tool being a collaborative resource that a number of understandings of rigidity and flexibility were centered. One particular understanding of rigidity was associated with this need to have other researchers inspect and sign off on contributed code. Harvey described this dynamic on some of the CDA's other production code:

> "So a lot of our other analysis codes are pretty heavily tested. And we hold ourselves to a pretty high development standard, with the branch-pull-merge architecture and you know, that sort of software development protocol, but that's, that takes a lot of people in the loop and which is why we do it. But if you're just trying to get plots moving quickly…" (Harvey, professor).

A high development standard here implies not only a certain amount of "protocol", and having people "in the loop." Specific steps that need to be followed in order to contribute one's code, but also the oversight of other researchers. As in the development of Pycosmo, regimented development required going through processes of code review, redesigning the code to address the needs of other stakeholders using the software, and providing documentation and tutorials that would help others use and edit the code.

This association with production code as a collaborative object–both in terms of who it was for and who needed to be consulted in its development–put it in tension with both a kind of autonomy and an understanding of speed of analysis work. Diana, another professor, similarly described the need to coordinate code changes with others:

> "But one thing about the passing around scripts that has been easy is flexibility. You know, I can modify it directly and I don't need to ask anybody's permission, and I can do things just immediately versus something like [pycosmo]. [...] And so it's been difficult from a developer point of view in terms of like, it doesn't have the flexibility that I need on a day-to-day basis to do everything, absolutely everything that I need. But that's also

> kind of like a pro at the same time, because, you know, they shouldn't let me just modify things, you know, it should go through unit testing" (Diana, Professor).

Diana here contrasts production code with flexibility, where flexibility means the ability to tailor new code to her specific needs but also the speed to do things "immediately." At the same time she designates production code as a place where she, just one specific researcher, should not be changing code arbitrarily. The implication in her description is that the code is a communal space, and uncareful changes could affect other users.

A critical point here is that this notion of flexibility is itself closely connected with the situation of unplannable, exploratory work. This notion of flexibility was repeated across many of my interlocutors in the field, and it reflected an understanding of the need to try things out quickly, in an iterative fashion, when one does not know what exactly the end point of the analysis should look like. Abe, a research scientist, described this as a kind of flailing:

> "I know for me personally, anytime I'm trying to solve a new problem I just want to get our hands on the data and start flailing around with it and, you know, just fail a lot of different ways until I get to something that sort of looks like it's working. I don't know about the industry side of things, but certainly on the research side of things, you want tools that are flexible, and, you know, give you the opportunity to fail fast, in some sense" (Abe, Research Scientist)

Abe's phrasing here reflects some self-deprecation and humorizing of his research work, but he is also trying to capture the situation of not knowing what to do next. "Flailing" reflects the activity of acting without a clear plan, of trying and failing. In this sense flexibility is a strategy that is adopted in response to uncertainty. However, it also entails being able to try things without much overhead of time or labor invested. Abe connected this with the use of Jupyter notebooks:

> "One of my observations is that the use of Jupyter notebooks has kind of exploded. We're doing kind of, you know, quick analysis and debugging of things because It's, you know, modular, it lets you change code on the fly without having to recompile the program and rerun everything. Like you can read data and have the memory change

> what you want to do, visualize it quickly with a plot, and then do something totally different. So I feel like, even the medium is different in terms of what you're trying to do." (Abe, Research Scientist)

The Jupyter notebook provides a coding environment in which blocks (or cells) of code can be run separately, such that one can try one variation of code quickly and then alter it and try it again. Here the notebook is closely connected with flexibility, meaning specifically the ability to try things without much time commitment and then backtrack to try something else.

Computational notebooks were in fact fairly closely associated with the notion of flexibility. Some members of the Radio Group worked primarily in Jupyter notebooks and used them to show their code and plots in lab meetings. The different usages of the computational notebook have been investigated elsewhere (Rule et al., 2018), but here I want to focus on how they create a space for a particular mode of working with code. Harvey, a professor, indicated this association in discussing the "nightly notebooks", a set of Jupyter notebooks in which members of the collaboration developed tests to be run on data coming off of the array each night:

> "The night notebook is interesting, it's kind of the wild west of the analysis, like all of our other analysis is pretty regimented and sort of carefully tested and, but the notebooks are just like 'throw a plot in there. Let's see what it looks like'" (Harvey, Professor).

Here again Harvey emphasizes the ability to try things and see what happens. The nightly notebooks are a space where members of the collaboration are examining emerging problems in the data collection process, and this involves writing code and producing plots where they are not sure what the outcome of the code will be and whether it will be useful in any longer timeframe. These activities constitute a kind of "wild west" in the sense that they are not following established rules for the development of software, such as those that exist around pycosmo. Work in the computational notebook is framed as being low overhead and rapid, allowing for the kind of exploratory work I described in the last chapter.

While both Diana and Harvey's quotations above indicate a desire for flexibility, they also both indicate a *need* for regimentation or protocol. In a positive valence, collaboration code represented the good qualities of reliability, robustness, and trustworthiness. The notion of flexibility could in fact take on a negative valence by contrast with collaboration code. Lucas, a professor, connected flexibility with "sloppy" coding:

> "My own philosophy is that research code is a different level than production code that does this- Our research is can you write an algorithm that does this better than any algorithm has done to date? Where this is a very particular step of the process. And you know, you have to try things out, you have to be flexible. You generally code in a sloppy fashion" (Lucas, Professor).

"Sloppy" or "hacky" were common terms for designating certain ways of working in the collaboration, where work was focused on immediate goals rather than robust design of software for long-term use. Some members of the Radio Group referenced a retraction that had occurred years earlier in the CDA, after which there had been an effort to change collective practice around software. This led, for instance, to a policy that code that was on the "critical path" to a publication had to have certain practices used in its development, such as code review and unit testing. In this way the processes associated with collaboration code were understood as a kind of rigor, whereas the processes of exploratory coding could be seen as a threat to rigor.

This notion of rigor in software development also tapped into a quite different understanding of flexibility. Thus far the associations I have discussed have fairly unilaterally aligned the virtues of flexibility with research code and the virtues of rigor and regimentation with production software. Production software was also discussed by my interlocutors as being useful for exploratory work precisely because of its robust and consistent design. Mason, for instance, stated that one of the benefits of pycosmo was that "...so it is nice to have a data container with lots of great attributes and methods that just kind of help you arrange your data right. In

scientifically useful ways.” By “container” here Mason is referring to the data structure at the center of pycosmo that organizes interferometric data and provides a number of useful functions for manipulating it. When I asked him what he meant by scientifically useful, he gave the example of particular low level data products produced by the WRT, which, in their raw form, were notoriously difficult to parse. Stella had done a great deal of work to figure out a consistent way to import this data format into pycosmo, where it became a much more usable resource:

> “So there is an order [to the raw data products], but it's like whatever the order was on the chip, when it got written. So [Stella] spent a lot of time figuring out exactly what that order was, so that she could read them into [pycosmo], where it was in a more user friendly order so that people could do different operations on it more quickly because the organization is clear. So I guess what [pycosmo] does is it… it brings a standard to organizing the data. It says you guys can write your files however you want. And if we support it with [pycosmo] we'll read it into this object. And this object will be standard no matter what telescope you're looking at” (Mason, Ph.D. Student).

Once the data has been read into pycosmo, it takes on a form that Mason is familiar with and which is consistent regardless of where the data came from. Mason uses the word “standard” here, but it does not imply the burdensome associations that some had towards standard development protocols. Rather, it designates a “user friendly order” which is ready to hand for exploratory data manipulation tasks. This was an understanding of flexibility that was produced precisely as an outcome of the processes of software production.

This understanding of flexibility was also a multidimensional one. It was in part a matter of the usability of the design of the software. Pycosmo meetings involved a great deal of discussion, for instance, about what were the most intuitive names for functions and what parameters should be provided to the user. In a deeper way, however, it reflected the fact that pycosmo presented modes of processing that were consistent, reliable, and for which researchers developed familiarity. As Mason indicates, standardization in the design of production code was a benefit rather than an obstacle to exploratory work. In this framing of flexibility, its opposite would be the slow, difficult process of doing conversions oneself.

Consistent processing also presented an alternative to risk. The pycosmo team called one part of the software package the “wild west” because it lacked some of the tests and guardrails for users that the rest of the code had. That part of the code, a feature called pyflagger, had been tacked onto the pycosmo package at an earlier time and never redesigned. During one meeting they discovered that the code was not running the standard “checks” of data objects before combining or manipulating them. This meant that in some cases its processes could fail silently, producing errors without the researcher knowing that something had gone wrong. This could cause confusion on the part of the researcher or even situations where they do not understand that their output is erroneous. This left some members of the group slightly aghast, and they agreed that Pyflagger was “the wild west.” This understanding of the wild west was slightly different from Harvey’s usage. It referred to a situation of risk, where software, if it was not designed carefully, might put computational analyses into places of error or confusion.

The checks being described here were implemented throughout pycosmo and prevented the user from doing things like combining data objects in ways that might compromise the outcomes of the analysis. As Mila describes some people (“black belts” at using the software) wanted to reduce or remove these checks, as they found them obstructive, but the developers maintained their stance on it:

> “[Others would say] ‘I know what I'm doing with my data. I want to do it. Don't error on me.’ And various things. It's like, no, that could be an error, because that is actually fundamentally a wrong thing to do. If you want to override it, you can go change the values or something and make it happen. But we want the casual user not to have surprising things happen to them. And so there are certain rigidities that are important to us about making sure that the data is- that everything's coherent and sensible” (Mila, research scientist).

Here Mila stands by the rigidities of production code as a benefit rather than a drawback. In particular these are rigidities in use, rather than in development process, and they separate pycosmo from a wild west where the user is free to do whatever they want, but the software

might run inconsistently or produce subtle bugs or errors.. Pycosmo was intended to be used as a piece of infrastructure for other kinds of work, and the surprises that Mila mentions here are not the generative kind that I discussed in the last chapter, but rather confounding breakdowns in code that should be held stable. This is discussed further in section 5.3.

	Exploration	Software production
Goal	Changing understandings, producing requirements	Producing robust software
Anomalies	Motive force	Obstruction, cause for switching to exploration
Planning and routine	Improvisation or variation in routine	Top down planning: specific milestones with rough deadlines

Table 2: Differences between the processes of software production and exploration.

Production code had particular temporalities not only in the speed with which changes could be made but also the foresight with which planning could be done. pycosmo meetings involved a great deal of long-term planning, in which members would create “milestones”, representing the fixes and feature changes they wanted to make over the next couple of months. Moreover, new code was contributed as if it would exist in the repository for a long time and be reused by many people. For instance, the development team avoided making large changes without warning. They used a deprecation process to warn other users that large changes, “breaking changes”, would be happening in the future, and they displayed deprecation messages for a long period of time before they actually made the change. During development discussions, the term “API change” was often deployed as a warning or caution against a particular design. An API change meant changing the keywords and methods that others used to interface with the mode, meaning that it would require changes in other software packages. API changes could be made, but members of the pycosmo team were leery of them, and they required significant warrant in terms of the new functionality that would be added. Stability, and reliability towards other researchers relying on the code, was a primary criterion. Collaboration code had different

temporalities in terms of planned permanence (Lee and Paine, 2015), the time range of planning activities (long term), and the time that was invested into each code contribution. Where research code had short-term planned permanence, and a rapid cadence of change to the code, collaboration code was changed slowly, planned over the long term, and was designed to operate on longer timescales.

Across different people's accounts, collaboration code and research code were complex categories, entertaining a variety of associations (Table 3). They differed in temporalities of planning, planned permanence, and rapidity of changes, with collaboration code engaging many of the problems of long term infrastructure development (Ribes and Finholt, 2009; Bietz and Lee, 2009). They also differed in *scale* (Lee and Paine, 2015), with collaboration code both requiring the input of a wider group of developers as well as consideration of an even wider group of stakeholders and their needs. The notions of flexibility and rigidity were understood in relation to these other aspects. Flexibility implied rapid changes without the overheads of long-term planning or designing for others, while rigidity implied moving more slowly, investing effort in the code as something that would be used for a long time by many people.

Research Code	Collaboration Code
"Not really regulated" "Get plots moving quickly" "The wild west of the analysis" "Hacky" "Flexible" "You code in a sloppy fashion" "You have to try things out" "One off" "Get the job done" "Fiddly" "Not robust"	"Software development protocol" "A lot of people in the loop" A "high standard" "Regimented" "Carefully tested" "Validated" "Bulletproof" "Rigidities"

Table 3: characterizations of research code and collaboration code.

There was ambiguity in the moral valences of these different ways of working. Flexibility was sometimes cast as essential to the research process, a quality of work in spaces such as scripts or computational notebooks that was necessary to “try things out.” It could also take on associations with sloppiness or “hacky” code, and the risk of errors or lost time. Similarly, collaboration code could be robust and reliable, essential for critical processes like publication analyses, but it was also seen as overhead or regimentation that obstructed autonomy and speed in investigating research problems. The distinction was also not a discrete one. During a discussion of research code at one of the Radio Group’s lab meetings, Mila pointed out, for instance, that there were design steps that could be taken even in the research process that would make it easier to later turn the code into collaboration code. In some cases researchers had hunches or ideas, even during the research process, of what kind of collaboration code they might later be able to produce.

Thus far I have discussed primarily how researchers in the Radio Group and the CDA *talk* about these different regimes of software work. That is an important dynamic, and while members of the CDA did not reference explicit development methodologies, their descriptions of these processes, mobilized between members of the collaboration, and via report to me in interviews, assert what these regimes *should be* and variations in those understandings. It also outlines how these categories fit into emerging understandings of rigor around software work, which will be discussed further in Chapter 7. However, these understandings or models of software process are bound up in practice and material enactments (Cohn et al., 2009), and it is the interaction of these things which constitutes software process for the CDA. Next I turn to specific practical activities by which these practices were separated from one another and tensions between them were resolved.

5.2 Repositories and Border Markers

The distinction between software production and exploratory programming was not just a mental model. It was enacted in the spaces of the Radio Group's work with software through boundary work (Gieryn, 1983), and particularly the mobilization of boundary negotiating artifacts (Lee, 2007). Boundary negotiating artifacts are artifacts mobilized in pushing and redefining boundaries, rather than just transiting those boundaries in a stable, routine fashion. For the Radio Group and its collaborators, the software repository, as a collection of interacting artifacts, became an important point of leverage for changing routines and practices around software development work. The repository in fact collects together a number of different artifacts associated with the work of software development, and these artifacts work in different ways (but in concert) to create a space for a particular kind of development practice. I will describe a number of these artifacts, including structuring artifacts (Lee, 2007), as well as what I call *border markers* and *mnemonic artifacts*.

The most apparent artifacts associated with the Github repository is its built-in issues and pull requests system. Issues provide a place for people to describe a problem and to discuss what should be done about it. It can also be labeled in certain ways to indicate what type of problem it is (a bug versus a feature request, for instance) among other things. Pull requests constitute a suggested contribution to the repository, defining the problem and requirements (often based on an issue), as well as what the contributor has done about it. It also provides a place to discuss and revise the contributed code before it is finally integrated into the repository. These tools can certainly be ignored on a Github repository, but on the pycosmo package they are used intensely. In particular they help constitute a process of planning. The issues define things that need to be done, and a place for hammering out the details of specific requirements. Moreover, the pycosmo developers often associated these issues with a milestone, a collection of issues that should be completed as part of a longer term goal of releasing a new version of the

software. The pycosmo team adhered to these tools closely, defining their work by picking issues from the backlog of available ones, and specifically those associated with the current milestone in progress. They also dedicated certain meetings entirely to planning work, where issues would be prioritized and added to the current or the next milestone. “Adding” an issue to a milestone is an action defined in the interface, involving navigating menus in order to add it to the list of issues under a milestone.

The pycosmo development team had also created templates for the issues and pull requests, which functioned as structuring artifacts to assert the necessary components of an issue or contribution. When one makes a contribution to the pycosmo repository they are confronted by a template, created by the developers, which requests particular pieces of information. These include a summary of the changes, a more detailed description of the changes, the motivation or rationale for the changes, an indication of the kind of changes (a bug fix versus a new feature, for instance), and a series of checklists where the contributor indicates that they have written tests or a tutorial or other requirements. Caleb, one of the developers, explained to me that these templates could be bypassed, but helped encourage people to carry out the necessary practices ahead of time:

> "So I have been contributing before we developed that template. And what we noticed is that sometimes people would make pull requests that didn't tell you what they did or things like that. And we know, as we want to expand to a larger possible developer community, we want people to be aware of the kind of expectations we have for their code. And that's where the template came from. And as I, you know, that's why it has this "I read the contributing guide," because a contributing guide outlines exactly how the rest of the managers expect you to prepare your code and or request. Of course, there are some people who just delete the template entirely and say I did a thing. Now you can't stop them, right? You never stop that. But it at least helps--you hope that some order will be instilled" (Caleb, research scientist)

The template can be deleted or ignored, as Caleb points out, but it attempts to enforce a material structure on the contribution, as well as the different components that a contribution

should have. In my own contributions to the pycosmo repository, the pull request template established specific obligations, empty sections of checklists, which I felt obliged to address. While details of contributions are often negotiated by the reviewers of the code, the template provides a preliminary structure for what a good contribution should look like, to which the contributor is obligated to respond. As Jack Goody has pointed out about the materiality of tables and checklists, "a table abhors a vacuum" (Goody, 1987, pg. 276; cited in Schmidt and Simonee, 1996). These tables or checklists might be simply written out textually, but they nevertheless enact a structure, through spacing and itemization.

Another structuring artifact on the pycosmo repository was more aggressive. Part way through my observations the group adopted a formatting tool that would format code contributions in an automated way. When one contributes new code to the repository, the formatter automatically reprocesses the code and changes the formatting to conform to a very specific standard for the Python programming language. This formatter could error and demand that the contributor make certain changes, but for the most part it took no input from the contributor and simply reformatted what they had written. Like the pull request template, this artifact helped assert a particular shape for the contributions to the repository, prior to any negotiations with code reviewers.

The pycosmo repository also included what I call *mnemonic artifacts,* which were used to maintain awareness of produced code without having to maintain complete knowledge of or attention to every software component at all times. These artifacts do not themselves remember or record information, but rather provide shortcuts for the attention of the researcher or developer. In the instances considered here they operate through a process of rapid automated checking, in the moment, but they function as an *aide-memoire* in that they allow the researcher to forget (see also Shankar, 2007), or simply to not hold in their mind the details of a large

opaque system. Among other things, they help make the ongoing practical development and use of a modular system possible.

A common type of mnemonic artifact is the unit test and the continuous integration (CI) tools that are a common part of the software engineering repertoire. On software like pycosmo, which used CI tools, new contributed code would be run against a suite of tests to make sure that the new addition did not break the functionality of the existing code. Specifically, the developers wrote unit tests which defined the proper outcomes of each particular function in the code given particular inputs. When presented with new changes, the continuous integration system would take the new version of the code and run all of the unit tests. If the changes that the contributor has made had interrupted, interfered with, or broken the functioning of some other part of the larger codebase then the test would ideally “break”, meaning that the expected outcome of the unit tests, given some established input, would not match the established gold standard. When this happens it notifies the developers on the Github webpage where the new changes are being discussed (Figure 11). The contributor is then required to fix any conflicts that have come up.

In some cases I watched developers of pycosmo remember, unaided, what conflicts some code they were writing would have elsewhere in the codebase. They have worked on many parts of the code and they are able to anticipate where these conflicts will be. However, the software is a myriad of interconnected parts, and it is impossible to remember all of the places that a given change might find conflict with existing code. Moreover, there are many parts of the code that they have not interacted with, and which they do not yet know will be affected. The mnemonic artifact allows the developer to recall the code that matters in a given instance, without having to keep it all in mind. In actual operation, the CI tool does this by running to code to see whether any unit tests break. Nevertheless, the artifact allows the researcher to forget the specific contents of the code, but to bring their attention back to the specific place where it is needed.

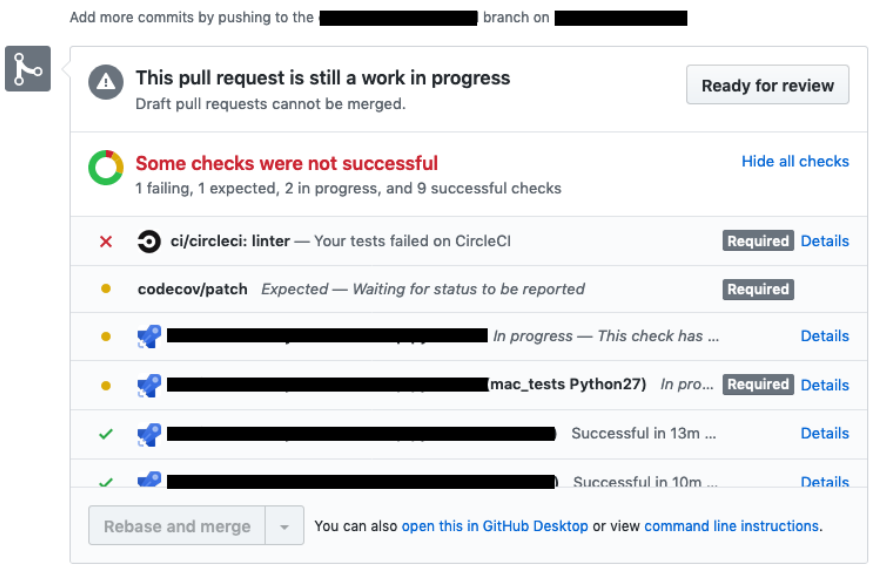

Figure 11: Continuous integration checks on the pycosmo repository block my pull request. This was a later stage when most of the tests were passing, but the system had previously required me to resolve conflicts with other parts of the code.

In 'reminding' the developer about specific lines of code, the CI tests are not just bringing particular bits of a software artifact to attention, but also the interdependencies in group and development efforts that hold around them. As described previously, the CDA collaboration had a large number of software tools that had come to be closely dependent on the pycosmo software. As Abe, a member of the CDA collaboration, described it, "[pycosmo] is the beating heart that runs through all of our [CDA] analysis." CI tests would resurface these interactions because members of the pycosmo development team had set up tests that would check new contributions against other projects being developed in the CDA. In the same way that the continuous integration system would run tests within the pycosmo codebase to establish whether something had broken, it would now retrieve (download) other software repositories that depended on pycosmo and run tests on that code that would establish that this new change in the pycosmo software had not broken "downstream" code in the CDA collaboration.

Over a messaging workspace, Mila described to me how that arrangement had emerged from a history of interactions between the development efforts:

> “Pycosmo’s responsibilities towards [the CDA] are a matter of ongoing discussion. Most of the pycosmo contributors and all of the maintainers are involved in [the CDA] at some level, and [the CDA] adopted pycosmo as a foundational library early on. We have agreed as a matter of courtesy to run continuous integration testing against some of their repos. Failures on those runs don’t necessarily prevent pycosmo's PRs from moving forward, but they do make us aware of potential issues so they can be addressed as early as possible. [...] But it came out of some early mistakes we made where we changed things without having a proper deprecation process and broke things downstream” (Mila, research scientist).

Mila describes an evolving relationship between the development efforts of pycosmo and the "downstream" development of other software in the CDA. The deployment of deprecation indicates a responsibility for the reliability of the software, that it should not change out from under “downstream” stakeholders. As Mila states here, however, it also asserts that the software will change, that it will adapt and engage new use cases with other stakeholders and members of the CDA will have to adapt to those changes. In this sense the external tests and the deprecation messages together help shape an interdependency and its mutual expectations. The software will change to meet the needs of other stakeholders, but it will do so slowly and with messaging to allow for “downstream” developers in the CDA to accommodate the changes.

These mnemonic artifacts (the external tests) are a material intervention which surfaces and resurfaces relationships between different researchers as a live concern. Similar to the way that Cohn (2019) describes the activity of pulling on a thread in the tangle of software to see what happens, the CI tests surface and make present particular interdependencies between groups of people. These are not only a matter of the interconnectedness of the code, but also of the histories of tensions and reconciliations between different stakeholders and development projects in the CDA. The obligations of pycosmo towards the CDA as a whole had been a point of tension in the past, because much of the CDA’s code relies on pycosmo, but the developers of pycosmo had previously tried to make it clear that pycosmo was intended to be a more

general tool, which would support stakeholders outside of the CDA. This had led to frictions in the collaboration, which were reconciled in part through the use of external testing and deprecation to make sure that pycosmo development would not have sudden, major changes on work in the CDA. The historical development of this interdependency in work, then, is tied up with the narrow technical characteristic of the downstream-ness of the software relationship. Moreover, the relationship can be reshaped by the deployment of external tests.

In the case of software systems, most mnemonic artifacts also obstruct, interrupt, or warn. CI systems, for instance, enforce awareness of particular parts of the code in a blunt way, raising bright red alert messages and blocking code contributions until the contributor rectifies their interdependencies with other code before they can proceed (Figure 11). Like with other aspects of the repository, these tests control action, and a large part of their rationale is to prevent the developer from overlooking or missing places where attention is needed. Mila expressed this purpose of unit tests in general:

> “If you change an interface to a function, you change the parameters that are taken in, if you have unit tests, you find out about that immediately, you fix all the relevant places. You don't have unit tests, like we don't have on [Software package], it all looks okay because you run a couple examples. You change it where you notice but you don't change it in the whole codebase. And six months later, a graduate student tries to do something slightly different than what you did and then their code errors and they don't know why or even worse, it doesn't error but it gives them some result that is subtly hard to understand. It's unclear if it's wrong or not. And maybe it's not even clear if it's different” (Mila).

Mila’s description of the problem in fact outlines a central difficulty of maintaining black boxed systems. It is always a concern whether errors or subtle influences are being introduced without the developer noticing or understanding. In these cases, then, the mnemonic artifact is not leveraged on command, but intercedes, alerting the researcher in moments when they do not even know that they are overlooking something.

Another kind of mnemonic artifact was the ‘check’ that was implemented in pycosmo. Many functions in pycosmo were written such that when the user performed particular operations, such as adding together two data objects, the software would check different aspects of the objects to ensure the integrity of the operation. For instance, my own contribution involved a relatively mundane adding task, but it had to make sure that a large number of parameters on the data objects, such as the number of antennas represented in the data or the timeframe covered by the data, were the same. In my case, based on conversation with the development group during a meeting, I wrote this as a fairly aggressive check that would raise an error (end the program) if these aspects of the objects did not match up. In other cases the developers decided to raise a warning that would appear in the output of the code. The pycosmo developers would often discuss the possible risks of accidentally performing a particular inappropriate operation. Users who were using the software for simulations, for instance, would sometimes want to override these checks and perform operations that would be problematic on real data. Like unit tests and CI systems, these mnemonic artifacts were focused on maintaining awareness of a complex and opaque system by directing attention.

Importantly, I do not understand the mnemonic artifact to take on meaning via reference to prior cultural meanings, as the term has been used elsewhere to to indicate (Ofosu, 2021). In my usage the mnemonic artifact is ‘dumb’ in the sense that it does not take on shape or meaning from prior action, nor does it represent or encode that prior action. The mnemonic artifact recalls, or calls up, the relevant parts of a complex, opaque system *at need*, such that one can mitigate the problems of forgetting or not knowing the extent of such a system. Like a kitchen timer, they allow one to forget, but then to attend to a situation when needed. In this sense mnemonic artifacts are prompts for the activity of infrastructural inversion. They indicate and locate breakdowns on the logic that it is better to be aware of disjunctures or conflicts between different aspects of a changing infrastructure than to have silent failures.

Lastly, the repository included what I call *border markers*. Border markers are artifacts incorporated into a repository that are externally visible, indicating the repository as a coherent entity, but also indicate the *laws of the land*, the routines and expected ways of working that hold when working within the repository. One of the first border markers for pycosmo is a set of indicators of the current state of the code. When landing on the pycosmo's Github page, one of the things that stands out first is a set of brightly colored badges. One shows that the repository's "coverage" is at 100%, meaning that every line of code in the repository is tested in one way or another by the repository's automated testing suite. Another badge indicates that all of these tests are currently passing, that there are no unaddressed breakdowns in the most recent version of the code. A third badge contains a URL to the Journal of Open Source Software (JOSS), where the software has undergone peer review and a summary of it has been published. Below these badges is a link to the software's documentation, and in the margin above them is a logo indicating that the repository has a code of conduct and a BSD-2-clause license.

Border markers are indicators that a software project is developed in an active and organized way, and that it has a certain level of commitment towards being a robust and reliable software tool. Malcolm, who joined the pycosmo part way through my observations, described this evaluation of the repository:

> "And I actually, I viewed the- may be the only person who's ever looked at unit testing and been excited. But I saw the kind of testing infrastructure that existed, and it kind of said to me, like, okay, this actually means that there's a certain baseline level of functionality that will always be expected in the software package. And so if I contribute something, like there is a hope, that the stuff will actually get maintained moving forward, like, it's okay, if this is like the contribution I make, and then I don't have to worry about the maintenance, because I'm trusting that the people working on the software package will help to carry that forward. It's just part of the normal day-to-day stuff" (Malcolm, Professor)

Border markers do not just indicate specific characteristics of a repository, that it has tests, but also a vague impression of the kind of work that goes on there, its adherence to development procedures and its likely sustainability.

One the one hand the border marker might be assumed to connote an intimidating or disciplinary presence, but they are not constrained to such a role. They can also be an inviting thing, and play a role in establishing coordinative interactions and fields of work. Malcolm, for instance, was a member of the HALO project and was attracted to joining pycosmo precisely because it indicated to him a certain amount of stability and sustainability. He had past frustrations with maintaining code in less formalized ways in the HALO project and came to the pycosmo project as a place to contribute that code such that it would hopefully become more maintainable over the long term. In this sense the border marker can play a role in synergizing (Bietz et al., 2010). As an artifact it can contribute to the establishment or extension of a field of work, and not just processes of coordination within an existing field of work.

In scientific working environments in particular, border markers such as these cut a strong distinction against a landscape of code maintained in other ways. Where the nightly notebooks were a "wild west" of analysis, the border markers attached to the pycosmo repository mark it as something else, a place where development is planned, designed, tested, and maintained over longer time spans. It is important to note that artifacts such as border markers are intentionally leveraged to turn a repository into such a space. Researchers in the collaboration, and particularly Ph.D. students, had repositories where they kept collections of scripts and plots that they had produced, but these were not intended to become collaboration code. In an industrial organization where certain development standards and practices are expected organization-wide, these markers might be taken as a matter of course. In the larger subfield of reionization cosmology these markers indicate pycosmo as a particular kind of place, in strong distinction

with other spaces where research work goes on. In this sense border markers indicate a way of working, and set expectations for people planning to use or contribute to the code.

Here I have outlined a couple of distinct boundary negotiating artifacts which operate around the pycosmo repository. These artifacts and their particular usage is not inherent to the Github repository itself. It is quite possible to use a repository without using these various features. However, in leveraging these artifacts the pycosmo developers are able to create a particular place for developing software. By place I mean that the repository, as an extended set of coherent interactions around a shared collection of artifacts, becomes imbued with particular meanings and expectations due to repeated patterns of use (Harrison and Dourish, 1996). There are a couple of different aspects to this. The pycosmo repository comes to be a place where development work involves long-term planning, and this planning is enacted through specific issues and milestones. It also becomes a place where development work is intentionally slowed down through artifacts like continuous integration tests that surface conflicts and force the resolution of distinct needs and interests. In this way the repository becomes a place where one must necessarily take others' use cases into account. Together these aspects of planning and slowing down development work enforce a particular temporality of collaboration code, a temporality that emerges from development that is "regimented" and has "a lot of people in the loop."

Looking at these various tools associated with software development work as boundary negotiating objects can help deepen our understanding of how software-related practices change in a research group, as well as how distinct contexts of work are established in an interorganizational scientific setting. The actual accomplishment of changes in practice in scientific labs and collaborations has been a central one in the literature on scientific software, and closer attention to the usage of the artifacts that *surround* scientific software (versioning systems, ticketing systems, testing tools, etc.) can help us better understand that process. In

this sense the concept of boundary negotiating artifacts can help us better understand specific empirical cases.

These findings also, however, extend the concepts of boundary negotiating artifacts in a couple ways. Most directly, they present new, specific boundary negotiating artifacts for consideration. Border markers and mnemonic artifacts, for instance, are specific ways of leveraging artifacts in pushing and negotiating boundaries. Mnemonic artifacts are not necessarily boundary negotiating artifacts in principle, as they can be used by an individual, similar to the way self-explanation artifacts (Lee, 2007) could be used individually or become a matter of coordination. However, in the context of the Radio Group's software repositories mnemonic artifacts such as the CI testing suite come to assert boundaries. In particular they force the contributor to make changes to the software repository with the needs and operations of a much wider set of stakeholders *in mind*. They surface and assert points of breakdown in the operation between designated sets of stakeholders, and in that way assert a development process that is regimented rather than autonomous with regards to making new changes. Examination of these specific artifacts, such as the continuous integration system, might be useful in other situations of scientific software development, but they might just as easily be useful in other cases of collaborative work.

These findings also extend the notion of boundary negotiating artifacts by extending our understanding of how they operate in groups. Lee (2007) mentions the possibility of examining constellations of artifacts that are leveraged around more or less complex projects, and examining how constellations might comprise both boundary objects and boundary negotiating objects. Other research in CSCW has also pointed to the importance of considering artifacts in terms of ecologies (Lyle et al., 2020). My findings point to ways in which artifacts can be leveraged together to create a *place* for a complex, multifaceted kind of practice. The kind of production work demarcated by the pycosmo repository in this case differed from exploratory

work in its temporalities, the standards it was expected to adhere to, and the protocols for input and oversight of contributions from collaborators. No one artifact asserts or cajoles all of these facets, but by leveraging them together a certain kind of space for development can emerge, even within a distributed collaboration like the CDA. While this gives an impression of what can be done with boundary negotiating artifacts in a constellation, it does not provide a more detailed account of changing topologies of artifacts and their usage around the repository. This is a topic that warrants further investigation.

These artifacts individually enforce different aspects of the development practice that the Radio Group were trying to integrate into their work, but when all of them are mobilized around the repository it is structured as a space where multiple dimensions of a process of software production are enforced. The repository takes on particular temporalities, such as longer term planning or building to requirements (required by structuring artifacts that connect pull requests to issues). They also enforce the scale (Lee and Paine, 2015) of the coordination to be undertaken as being large in the range of stakeholders rather than being a small autonomous project. They also enforce particular standardized forms for writing and documenting the software. I have described this as creating a particular “space” for software development work (one of production rather than exploration), but what I mean by the term space here is more like a field of work (Schmidt, 1994): it establishes *and regulates* a coordinative space where actions are made perceptible and interactable through a variety of artifacts.

It is in looking across these artifacts as they work in conjunction, that we can begin to answer the question of how researchers accomplish the integration of planning and design work into their research work (my third research question). The contributor, especially the new contributor such as myself, encounters these systems together. Creating a pull request and contributing code requires interacting with all of the artifacts I have mentioned here, moving from border markers to structuring artifacts in the creation and contribution of code to navigating tests in the

automated evaluation of contribution (the CI process). The primary development team on a repository can guide and shape action through the concerted mobilization of multiple artifacts (Latour, 1992).

The use of these kinds of artifacts (unit testing badges, pull request templates, and continuous integration systems) are of course ubiquitous in contexts of industrial software production. What is worth observing here, however, is how they are leveraged as coordinative artifacts in order to materially delineate kinds and patterns of practice, in this case between exploration and software production. This is particularly significant in the situation of the scientific community, which must simultaneously maintain multiple, quite different kinds of processes of software work. I will return to this issue of how change is accomplished in development work in Chapter 7.

5.3 Robust code, robust phenomena

In section 5.1 I described an understanding of flexibility in which production code was understood to support flexible, exploratory work precisely because, and not in spite of, its standardized character. This dynamic is a central one for the research software system and it is a virtue of production code for scientific work that has not been well outlined. The benefits typically associated with implementing software engineering techniques in the sciences focus on issues of sustainability or robustness. These include that robust software is easier to pass on to others (Hong, 2014; Marshall et al., 2010), that it is more sustainable and requires less work in the long run (Carver et al., 2021; Ram et al., 2018). There are also some arguments along epistemic lines, that software engineering practices will produce software that is less prone to error or to lead to retractions (Wilson, 2014). This latter is close to the issue that I want to approach here, but it is typically focused on the benefits of production code for checking or demonstrating activities, such as journal review or post-publication replication. There is an

important sense, however, in which production code contributes to the *exploratory* process. What I mean by this is that in reorganizing around software production researchers are able to gain new leverage in the process of probing and elaborating poorly understood phenomena.

In order to examine this dynamic, we must look at how researchers build confidence in their interpretations of outcomes of exploratory tests. In running a test that produces some nascent phenomenon, such as the “evil cow”, researchers in the Radio Group run different types of code. Some of this code is research code, in the group’s typical designation, which the researcher wrote in a tailored way for the purposes of that specific test. This code also, however, invariably draws on a great deal of other production software packages, from the plotting libraries the researcher might be using to low level mathematics packages to data manipulation or conversion packages like pycosmo. This is what Howison and Herbsleb (2010) mean by considering software as an ensemble artifact, that it is a great many component parts operating together to perform a singular test or analysis, sometimes to produce a single plot.

Importantly, this software varies not only in how it was produced, but also on the type of epistemic relationships that the researcher has towards these different kinds of code. Some of this code they have confidence in because they wrote it themselves, they are still familiar with it, and they know exactly what it does. They might have confidence in other software components, particularly a great deal of the production code that they draw on, because it has been used by many people and there are processes in place to validate and check it against established tests (e.g. unit tests, among other kinds). This is what Howison and Herbsleb (2010) describe as personal and external logics of correctness, respectively. This complicates the sense in which the researcher “knows” their instrument. Paine (2016) has previously described the importance

to researchers of knowing an instrument “in your bones”, and this sense of familiarity with the tools one uses is often discussed as a virtue of good research and good researchers.[3]

This understanding of one’s infrastructure comes to matter directly in the interpretation of exploratory tests. Small mismatches in standards or small errors in any of this underlying code affect the nature of the phenomena that a given test brings into being. Harvey, a professor in the CDA, described a particularly hairy example of this problem. He was attempting to compare different pipelines that had been developed within the CDA and another EoR-focused collaboration by switching out different analysis components from the two pipelines. The output of the two pipelines he ran should have been at least similar, because they were processing the same data, but the final outputs were not looking similar at all:

> "So we were looking at just the XX images, they just weren't the same. There were blobs here and there's no blob there. There's a blob there, there's kind of the same… but just not. They're not the same. [...] So we take these- I take these differences and I just get more shit everywhere, and it was really confusing, and really confusing and really frustrating" (Harvey, Professor).

Harvey said that he worked on this problem for months before realizing, by accident, that the conventions used to represent the polarization (directionality) of the antennas in the array was different between the two software pipelines:

> "And then I made a mistake one day and accidentally flipped the indices in my polarizations, in one of them, and they look the same. And at first I thought I made a mistake and loaded the same image twice or something... but no there's a little bit of difference, but not nearly as much. so on a telecon I say um, is this- [sounds indicating bewilderment]. [...] I said [Colleague] what polarization is XX? And he goes, Well, it's the polarization aligned with zero angle at this latitude, like all radio astronomy. And, and I say You know, [Mila] or whoever, what is polarization X for you? And they say, well, it's the East/West, you know, it's the zero angle. So we literally have just a 90 degree rotation... took me six months to find it. About standard for me" (Harvey, Professor).

[3] McCray (2004; cited in Baneke, 2023) mentions this in the context of large astronomical instruments in particular.

Many of the members of the CDA had stories like this, of having lost time and been misdirected or confused by breakdowns in the quotidian operations of format standard conversion or file reading. Indeed, hunting down small bugs in underlying processing code was a regular part of the work of students and research scientists in the Radio Group.

Harvey's case highlights the fact that in interpreting the outcome of a test, particularly an open-ended one, one is not evaluating the action of a single piece of code. Rather they are evaluating a vast network of operations in various software components, hardware components, and beyond, including interactions the researchers estimate might be happening in the earth's ionosphere or at TV broadcast stations near the array. The interpretation of any nascent phenomenon might be undermined by poorly performed data manipulations or due to the characteristics of the cables used on site. Moreover, judging the behavior of a phenomenon between iterative tests in an exploratory process means being able to rely on consistency in the operation of underlying components between tests. Whether this massive infrastructure of instrumentation is working as intended cannot easily be deduced from the phenomena that are produced themselves, as there are no gold standards (no oracles) for outputs in open-ended tests.

The researcher's instrument is in this sense engaged in what Rheinberger (1992a) calls a "non-trivial" interaction between instrument and phenomenon. In most cases, the majority of the group's software has defined gold standards, embodied in the unit tests described in section 5.2, and their proper operation is known. In the open ended test, however, this code is remobilized as an ensemble with new components, including potentially new data, but also new code that the researcher has written to vary the functioning of this larger infrastructure in some way. This variation of the operations of this larger codebase is aimed at probing some characteristically unfinished entity, such as the "evil cow."

One outcome of this is that establishing the character of nascent phenomena (and eventually one's diagnosis of them) was a matter of building confidence and familiarity with the instrumental ensemble, rather than any kind of mechanistic proof.[4] The Danielle described what this problem looks like in practice:

> "But anytime you find something interesting, there's really no way to know, is that something interesting scientifically? Did I just have a bad understanding and it's actually not interesting, or is there a bug in the code? You have no idea. And so it can take a while to answer that question because in order to figure out if there's a bug in the code, you first have to figure out on a deep level what is supposed to be happening, which can take a while. [...] And so figuring out whether something is doing what it's supposed to do is a less trivial question than you might expect. So it can take a while and you want to be sure, but it's a hard thing to be sure about because there's not necessarily a right answer a lot of the time. And so it's like I want to have confidence in what I'm making and that takes a long time to build" (Danielle, Ph.D. student).

Danielle is describing a point in the midst of the iterative exploratory process I described in Chapter 4, where phenomena are understood only in terms of their immediate characteristics (e.g. the "one third line") and it is still unclear what part of a vast infrastructure might explain their character and appearance. The process of working through this is not a matter of one definitive test, but of building confidence in one's understanding of the way this large ensemble is working. In Harvey's case, he lost months of time trying to make sense of outputs that had more to do with the way his code was handling polarization conventions than with any potential astrophysical phenomena, simulated or not.

The development of productionized software, or collaboration code, helps with problems of the type that Harvey has encountered because they help build continuity in the "technological conditions" or "identity conditions" (Rheinberger, 1992) that define a phenomenon. A research

[4] This situation of instrumental uncertainty is also part of the basis for Collins' (1992) idea of the experimenters' regress, and he makes a point similar to this that resolving these kinds of bootstrapping problems between instrument and outcomes is a matter of enculturation into particular communities. My own explanation will go by way of the notion of Rheinberger's (1992a) notion of technical conditions or technological objects.

software system needs to be able to achieve consistency in its reproduction of phenomena for an object to take form across sequential tests, and tests between groups or people. In providing consistency in these identity conditions, a research software system contributes to the closure of the experimental situation. If the various modular packages leveraged in a test can be relied upon to be working as intended, to perform the proper transition from input to output, then that closes down the number of possible understandings we might have for the character of a phenomenon. If one iterates on a test, they can be more confident that any differences were a result of changes made, rather than some irregularity in a data manipulation package that is being leveraged as a dependency. In other words it contributes to the *ceteris paribus* rationale of testing or experiment (Pinch, 1993), in which the object of investigation gains coherence through reproduction based on the idea that everything else has been held the same.

This contributes to coherence not only between successive tests, but also between researchers. As Stella pointed out about the use of pycosmo in the CDA, the use of standard, codified processing software (which for the most part performs the same processing on different computers) contributes to the comparability of phenomenon between researchers:

> "...I think that consistency is really important. So having lots of people using the same tool, again, just makes it a better apples to apples comparison when you're looking at what results you get, knowing that you've got them in the same way is really reassuring." (Danielle, Ph.D. Student)

In the effort to begin developing an understanding of what a phenomenon is, the ability to get it "in the same way" again and again is a critical aspect of coherence. Even in a process of differential reproduction, where difference or variation is pursued, establishing continuities between successive tests is a necessity. Robustness and reliability in one's software infrastructure contributes to the robustness of their phenomena across time and between people.

In the particular context of software systems the effort to maintain a large complex infrastructure as stable technical conditions for the conduct of exploratory work becomes in part a game of modularity. What I mean by this is that the dynamic of modularity, which is embedded in many software engineering practices and recommendations, becomes a primary way of trying to maintain stability across a complex instrumental system. The original understanding of modularity pertained to what a developer needs to know about a complex piece of software. If standard specifications of proper input and output can be asserted between modules then the developer can work on one part of a system without needing to understand or make changes to most of this complex system (Baldwin and Clark, 2000). In many classical cognitive models of modularity the construction of modules allows for the offloading of thinking, it is like the automation of a routine that allows for the mindless performance of a task (Ashforth and Fried, 1988; Schank and Abelson, 2013). The development of robust modules in scientific software contributes to an iterative testing process under a similar logic: it allows the researcher to isolate causes and interactions by attempting to hold stable a large part of infrastructure they draw on to constitute a phenomenon. This does align to a certain extent with what I mean by the notion of a contractual relationship with software, that the code has well-defined requirements and one takes them as given.

This notion of modularity places a great deal of emphasis on a kind of design decision, and does not account for the relational maintenance work (Bietz et al., 2012) needed to maintain infrastructural systems in an ongoing way. In order to better characterize this we can consider the construction of infrastructural software as a process of delegation (Ribes et al., 2013). Through delegation the researcher can certainly take their mind off of the procedures intended to be performed by the software, at least to some extent, but the delegation is not unproblematic. It produces a relation between the researcher and the delegate, which can degrade or break down and must be re-established in an ongoing way. If we look closely at

modules we can see that they occasionally break down, and that they are maintained as working pieces of a larger system through a larger network of actors. Stella made this network visible when I asked her what she meant when she referred to pycosmo as a reliable software package:

> Will:
>
> “And you said it was reliable. Does that have to do with- what does that mean?”
>
> Stella:
>
> “To me, that means that it usually works. It usually works. And if it's not working, there's a lot of support for that. So if something isn't working, I know that I can create a PR and somebody's gonna fix it, you know. Or maybe I'll end up fixing it maybe [laughing], but that it's fixable. [...] But yeah, it's, it's something that feels- I don't feel worried about it. Or if I don't understand what it's doing I can look at the code and I can usually tell like, ‘Oh, this is how it's doing that’ or ‘this is what's going on’ and so I know what's happening if I feel like I need to.”

This quote captures the reality of the delegation to a module of production level code. Stella’s understanding of reliability is not a blind trust in the correctness of the code for all instances. Nor is a matter of knowing the code through and through herself. Rather the code is something that “usually works” but that can vacillate between working order and uncertainty. The reliability is in the scaffolding of this ability to reinspect and reinvestigate the code, and to bring it back into a state of working order. Of particular note in this is the presence of maintainers working on the software. When we examine the reliability of the software artifact we find behind it the ongoing work of human actors.

Another point embedded in Stella’s description is that collaboration code sometimes broke down in new circumstances, and researchers would need to approach it anew as something that needed to be worked on and investigated. This was of course the situation in Stella’s initial flagging problem, described in section 4.1. In another instance, Stella discovered that the

numbers for antennas on the array was being handled differently in pycosmo than it was in their primary analysis software, meaning that data passed back and forth between the two could have antennas mislabeled. Stella then had to make changes to the pycosmo software to change the assumptions that the software made about antenna labels.

Most often, this happened when new stakeholders joined a project or when users started new projects with new goals. For instance, one member of the pycosmo team began performing simulations for positioning an interferometer on the moon, which required pycosmo to overhaul its system for rendering telescope positions. In another major case, a new interferometer project, called the High Altitude Low frequency Observatory (HALO) joined the pycosmo project. One of their members, Malcolm, had to make significant changes to the software in order to make it work for this new instrument. This upended a large number of basic assumptions in the software, and in the course of doing this Malcolm discovered that the coordinate calculations that pycosmo was performing were off in certain cases by a few arcseconds. In response to this Malcolm compared the coordinate calculation code for a number of different libraries and discovered that a common astronomical software package that pycosmo was using was slightly off when projecting coordinates into a specific coordinate system. The issue that Malcolm created for this laid out all of the tests that he had done, including a plot of the comparison of the accuracies of the different systems. In this investigation Malcolm is running the various software tools once again in an open-ended way, to see what will happen and to produce unexpected results.

The point in these cases is that productionized software does not always stay productionized, but rather it too can come under new scrutiny. The process of software production sometimes reverted into exploratory work. To be specific, when major breakdowns of these kinds occur, at least at the outset there is once again uncertainty about what it is that the code should be doing. In many cases the broader requirements for the code may not ultimately change, but the code

temporarily falls out of a contractual relationship with the researcher, and it must once again be run in an open-ended way. Moreover, researchers' usage of productionized software was not guided by blind trust. It was more a matter of confidence, which could be shaken or reinforced. This is the moving target of infrastructure (Star and Bowker, 2004), and it means that researchers changed their orientation towards different parts of their software (or even towards broader open source software packages) at different times.

How does this change our understanding of the effort to build robust scientific software? Latour's (1987) usage of the black box metaphor provides a useful perspective on this. Latour marks a transition between the effort to establish and demonstrate the working order of a system, and the usage of a system on the basis of its working order. In the exploratory work described in Chapter 4, the working order of particular pieces of software is what is at issue, and what the labor of the researcher is intended to establish. Such software "cannot convince anyone", as Latour puts it (1987). Production software is a reversal of this dynamic, such that it is the understood working order of the software that contributes to the coherence of the phenomenon of interest. The researcher can be more confident in the phenomenon because the software is taken to be in working order. While research code is under examination and its working-ness is at issue, collaboration code is ideally an authority.

The larger point of these arguments is to specify how the development of robust research software contributes back to the epistemic work of generating new phenomena and new capacities. In my characterization it does this by bringing consistent and reliable actors back into the instrumental system, which helps that system become a technical object in Rheinberger's (1992), which is characteristically determined. For the research software system, this situation of being characteristically determined was not a matter of just knowing all parts of an extensive codebase, nor was it a matter of defining modules and forgetting about them. Rather it was an ongoing activity of maintaining the working order of different modular systems and managing

new relations that developed between them and new stakeholders, new datasets, and new software resources.

5.4 Research software systems

This concept of the research software system helps us understand how the Radio Group and their larger collaboration accomplished the maintenance of software over time as research problems and stakeholders change. Specifically it answers this by way of coordination: they established distinct regimes of work in order to designate a place for software production work. This approach is not, for instance, the unilateral application of best practices for software engineering. Scholarly discussion has outlined a number of good qualities for research software, including robustness and sustainability, as virtues of a monolithic category of software (Wilson et al., 2017). However, the CDA maintains two distinct regimes of software work, where different virtues or criteria for development work are recognized. Some of the researchers' characterizations of the regime of research code are pejorative, such as research code being "hacky", but their larger goal is not to remove the regime of research code entirely. Spaces for exploratory programming and writing research code are maintained as places with low coordinative overhead and short-term planning temporalities so that researchers can "get plots moving quickly" and "try things out." If we take the exploratory mode of research programming seriously as an essential process in scientific work, but also recognize its interconnection and embeddedness in processes of robust software production, then we must consider a *research software system*, with heterogeneous parts, rather than scientific software as a unitary category with a singular set of criteria.

The notion of the research software system is not a prescriptive model of a simple binary system. A given collaboration or research group may distinguish different understandings of what contexts count as exploratory and how they relate to specific other contexts that count as

production. What is essential about the research software system is the effort to distinguish different processes of work in order to reconcile the needs for flexibility and rigidity in software systems. It is an organizational method for accomplishing flexible yet coherent and consistent research tools in an ongoing way. In this sense it is an effort to resolve, in the specific context of developing and using software tools[5], tensions that are common to research infrastructures (Bietz et al., 2012; Ribes and Polk, 2015). The Radio Group has done this through the distinction of two roughly defined regimes which have some overlaps and which have their own internal heterogeneity. There are, for instance, different kinds of activities that fall under exploratory work, such as group programming in notebooks over a call or individuals trying things out in personal scripts. Members of the Radio Group and the CDA know this, and do a great deal of boundary work to determine whether something should be worked on in an exploratory way or in the mode of software production (see section 6.2). They nevertheless discuss and use rough categories of production code and research code.

Another significant aspect of this heterogeneity is that what we consider 'good' research software development cannot be limited to a single set of characteristics (robustness, sustainability, etc.), but rather it must consider distinct kinds of work on software and the coordination of these different kinds of work. Baker and Bowker (2007) have made a similar point about memory in information ecologies, that it is enacted in different ways and that these different modes for memory interoperate. Similarly, scientific software is enacted in different regimes, and these regimes must interoperate. Bietz and Lee (2009) have also observed the heterogeneity of databases which are pulled together in scientific work, despite researchers'

[5] This situation of researchers developing their own software is often described as end-user development. End-user development or software engineering is usually designated as the situation where a person is developing software for their own use, and conversely they are the primary intended user of the software they write (Ko et al., 2011). In a number of ways the Radio Group departs from this category because they develop code for their collaborators. However it is important to note that this mode of managing flexibility and rigidity in infrastructure is one specific to this situation where the researchers are also building the software, rather than having, for instance, an external company do it.

references to a singular "Database" (pg. 2). Scientific software pipelines, too, are often referenced in the singular but enacted in the plural, with multiple different kinds of software needing to be pulled together to make an analysis go well.

The notion of the research software system not only asserts heterogeneity, but also points out attention to the articulation of these distinct kinds of work. In describing an effort to ‘maintain the technological conditions’ I characterized one essential way that the work of software production contributes to the work of exploration. This is, in part, the local articulation work of making software tools robust and usable for their application in future exploratory endeavors. In the next chapter I will outline a process of graduating code, by which exploration contributes to the work of software production. A key aspect of this interaction is the graduation of code, the transition of a code from the context of exploration to some specific repository (or a new repository) where it enters a context of production. Determining how to do this, when it should happen and where it should go and what specific practices should be applied to it, is a nontrivial kind of metawork (Gerson, 2008).

The move towards consideration of research software systems has implications for how the social studies of science, scientists, and science policy makers might better support and plan the work of research software infrastructure development. In particular it moves our attention from the implementation of practices for writing and designing software towards the scaffolding of the articulation work (Schmidt and Bannon, 1992) that enables heterogeneous research software systems to work smoothly. Importantly, this is not repudiation of the adaptation of software engineering knowledge for the sciences. While much can be learned about supporting coordination from the field of CSCW, the techniques and understandings of the field of software engineering have always been matters of organizing work and not just writing code. This issue that surrounds current engagements between software engineering and scientific work is that in

the context of ‘best practices’ it has been framed as being minimal and codified (Wilson, 2006a; Wilson, 2006b), in part as a way to minimize the overhead of adopting new practices.

Based on the case I have developed here, I would argue for engagements with software engineering knowledge which are expansive rather than minimal, and adaptive rather than codified. By expansive I mean, for instance, that researchers might move past the minimal and most accessible techniques of software design for a deeper engagement with software engineering methodology and more abstruse techniques of the profession. By adaptive I mean that these techniques are not best engaged as pre-packaged black boxes for software work, but rather as resources that scientists can freely adapt to their own problematics. An example of both of these characteristics is the example of the use of deprecation described above. The members of the pycosmo team take up the practice of deprecation (which is not included on most lists of best practices) and use it to re-negotiate the responsibilities that hold between development groups in the collaboration. This is a deeper engagement with one of the more abstruse techniques of software engineering, but it is also adaptive of that technique to the specific coordinative tensions of the CDA.

Chapter 6:
Code growth and the development of novel capacities

The research software system is a system for producing novelty, and an important object that this makes available to us is dynamics of development in the system as a whole. This is not the primary objective of this study, but I will engage it here both to demonstrate it as a space for future work, and to fill out certain aspects of my answer to my second research question: how the Radio Group maintains software over time as problems and stakeholders change.

The immediate question is what exactly is developed in the development of a research software system. My answer to this is that it is capacities for engaging and interpreting phenomena that develops. However, this occurs hand-in-hand with the development of different kinds of software. Prior work has looked at how software in open source contexts builds up, how new functionality comes to be added to existing functionality (Howison and Crowston, 2014). It is worth asking a similar question about scientific software, not only about the robust repositories of well-tested code but also about the large amount of research code that is produced in the research process. In section 6.1 I will focus on the latter. I will discuss the tendency towards growth of research code, as well as the notion of *residual code*, which occupies a grey area between code that is dispensable and code that will 'graduate' to become robust collaboration code. This will lead us into discussion in section 6.2 about this process of graduation itself, both as a transition between ways of working with software and as connected with understandings of rigor, but also as a point of tension in the organization of a research software system. Last, in section 6.3, I will discuss capacities, what they are and how they build up. I will further discuss how the examination of capacities can bolster certain approaches in science studies.

6.1 Code Growth

One way to understand how the research software system proceeds through time is to look at the accumulation and maintenance of lines of code. As I will discuss in section 6.3, this may not be our preferred measure because it is centrally focused on how the software changes, potentially framing the creation of a software product as the primary goal of the research software system. Nevertheless it is a useful thing to consider because the development of the software artifact itself is essential to the development of novel capacities and because it gives us insight into how researchers extend and update their instrumentation. Here I will argue that the exploratory programming process has a tendency to generate a great deal of code without clear stopping points or end points. Moreover, researchers maintain this code through somewhat bifurcated modes of preservational versus revisionist habits of maintenance.

Research code, which was produced in the unpredictable and winding process of exploration, had a tendency to pile up. As described previously this work was oriented around epistemic objects (Rheinberger, 1992a; Ewenstein and Whyte, 2009), which precipitate unfolding or alteration. For instance, long-running research projects center around objects like the “evil cow”, which are materially enacted in the group's instrumentation, but prompt or raise poorly formed questions. They help project work by shaping curiosities and unknowns. Additionally the investigation of problems around these objects often raises new problems, which also demand investigation. Working in this context, researchers move easily from iteration to iteration, pursuing the unfolding character of their research problems without finding clear stopping points. Danielle described how this dynamic could become a problem for the process of graduation:

> "...but then I think part of what happens is you run into problems where you start a piece of code that it's just supposed to do something small, and then it slowly grows and grows and becomes something really useful. And then all of a sudden other people want to use it, and then they're adding to it and then it becomes like this messy, messy thing that you wish you've done better, but you didn't realize how useful it would wind up being or that other people would ever see it. And then it's really hard to go back and retrofit it. And I think that's probably exactly what happened with [Analysis software], was that no one expected it to be as good as it was? And then once it was, everyone wanted to use it and it wasn't built for other people to use it" (Danielle, Ph.D. Student).

Danielle’s point here is connected with the improvisational nature of research work. Because research work does not proceed by plans, it does not project ahead what the proper form of the

software should be. Rather this emerges in trials from one iteration to another as researchers gain a better understanding of their problem. Moreover, there are not good stopping points in this process, as Danielle indicates. As indicated by the notion of the *indefinite rhythm,* discussed in Chapter 4, research work did not have distinct end points, but simply proceeded from one problem to the next in a regular cadence of work.

Within this improvisational mode of work, research code takes on some strange temporalities. In the midst of investigating a problem, the researcher cannot have a strong sense of what code might end up being more broadly useful. For this reason all of the code they create has low planned permanence (Lee and Paine, 2015). However, research code is often something that is kept around for quite a while after its creation, because group members often wanted to keep track of what had been done at past steps in their analysis, or they may want to run that test again for a new problem. This created a phenomenon of *residual code*, which is code that inhabits a grey area between deletion and a transition to production code. Although not useful enough to turn into robust software that must be maintained, it provides promise for potential future use, or simply reference, which warrants keeping it around.

This residuality became visible by contrast with a process of graduation, where code would proceed into a process of being productionized. During an interview Noah showed me a script where he had organized a kind of history of his past analytical work under different if-statements (Figure 12). This function structure provides an informal trace of the exploratory process and its transition to production code. The first clause indicates a kind of baseline, the established cosmo-rfi approach that Mason had developed, while the second two clauses represented the ‘imaging’ and ‘uv plane’ approaches that Noah had taken, as discussed in Chapter 4. As he had proceeded on to new techniques, he had put this code behind an if-statement as a way of “commenting it out.” ‘Commenting out’ is a frequent practice in programming where lines of code can be marked as comments, such that they won’t run when the program is executed, but

they can still be read by the programmer. This is typically supposed to be used by messages aimed at other humans, but it is often used to suspend or shelve certain lines of code that one does not want to run but which might be needed again. Noah's somewhat irregular use of the if statement performed a similar function. The if statement makes it such that Noah could provide different keywords that would trigger code written for different testing approaches (1, 2, 3, or 4 in Figure 12). At the point of our conversation he was not using this code from the earlier tests, and in fact admitted that it probably would no longer work because he had changed things elsewhere. This informal design, however, holds that code in abeyance: it is present and could be picked back up if Noah had to return to that approach for some reason, but it does not run unless explicitly called. It is not an essential or working part of the larger codebase at this moment. Code retained in this way could also be inspected again, to remind oneself of what they had done in a previous iteration. Because of the uncertain path of exploratory work, it was never entirely clear when one might be totally done with particular code. When I asked if they would ever come back to that code and reuse it, Noah was uncertain. He said that he or a future student might come back to that approach. It was hard to know. Noah was keeping the code around just in case he needed it.

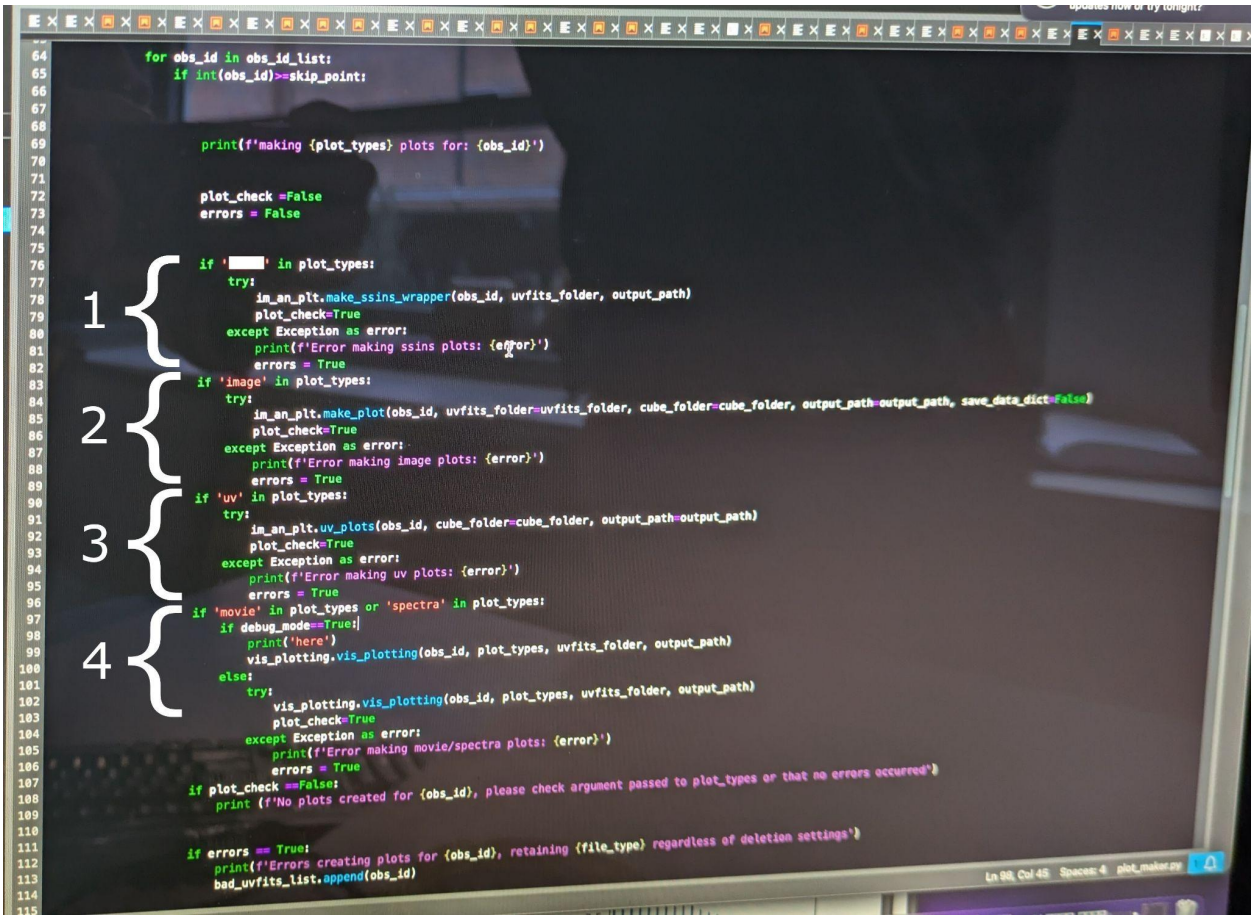

Figure 12: One of Noah's scripts, where he has organized different analytical approaches under different if-clauses. The clauses marked by the 'image' and 'uv' variables were older approaches, which no longer worked. The clause identified by 'movie' or 'spectra' contained code that Noah planned to contribute to a repository.

By contrast, the last if statement in Noah's script represented his most recent approach which had produced the "evil cow" and "stratospheric clouds" phenomena, among others. This had turned out to be a useful algorithm, due to its effectiveness in detecting these new kinds of RFI, and Noah was planning to contribute that particular part of the code back to the cosmo-rfi repository, to be used in a more permanent way. It is this way that certain code comes to be selected out of the accumulation of research code and moved into a production context. This evaluation is made based on the perceived usefulness to others or on how frequently it will need to be used in general, but the judgement is typically made in retrospect. More senior members of the group often had a broad vision about how the goal of a project such as Noah's might

result in a useful, general tool, but specificity in code and technique was only reached through iterative development.

In the Radio Group, plotting code had a particular tendency to become residual. Plotting code was code used to create and configure the plots and visualizations that were at the center of the Radio Group's exploratory work. There were certain categories of plots that were frequently used in the group's work, such as those showing RFI features in low level data products, but often these required a high degree of tailoring to specific research questions at hand. Stella described the situation this created for maintaining or saving this code:

> "And I know there's been a couple of plots that I have generated recently when [Magnus] and [Mila] have been like, 'oh, this is a plot that we should make. Where does that code live?' We don't actually have a place of 'here are these 20 plots that we would like to really see or that we found really useful, and here's kind of the code that you can use to build them.' So that's something that I'm noticing now because they're like, 'oh, we don't want to lose this plotting code that you've written.' And I'm like, okay, but what do you want me to do with it? That's a little- so that's something that I think isn't contributed as much, or we're still trying to figure out how to preserve that. And we did recently, in the last couple of years, create a repository for student projects, and my plan is to take the various Jupyter notebooks that I've used to do stuff and just throw them in there and be like, 'good luck'" (Stella).

Stella recounts the more senior members of the lab being interested in maintaining this plotting code that Stella has built up as a kind of organizational knowledge. From my own perspective, Ph.D. students in the Radio Group had significant capability for designing and programming plots, and it constituted one of the primary skills necessary for work in the lab. The ability for the group to make sense of some emergent phenomenon depended on a matter as simple as a color bar. Magnus often requested highly complex visualizations, involving multiple plots aligned in time or in frequency, metadata or labels indicating categorizations or IDs of the data, and highly specific color bars and axes.

Given the work that went into the production of these plots it made sense to understand them as a resource that could be leveraged again instead of reinvented by new incoming students. However, as Stella indicates, there was no explicit method in the lab for doing this. Elsewhere in the collaboration they kept exploratory plotting code in repositories of computational notebooks with chronological titles, such that the code was still available, if not easily navigable. While walking through some of these notebooks, Harvey pointed out to me that it is very difficult to test plotting code, and so it was often not 'graduated' to be production code. This, combined with the fact that it was typically tailored to specific cases, made plotting code a particularly difficult kind of resource to maintain over time, but one that was essential to the exploratory process. It was essential both as a record of results but also of kinds of strategies that might be reused.

The notion of residual code highlights a dynamic of divergent rationales in the recordkeeping around code in the collaboration. Shankar (2007) has highlighted the importance of how scientists manage the huge accretion of notes, results, and data in order to create a functional record, particularly through activities of synthesis and selection. Cases such as Noah's demonstrate a divergence in the Radio Group around the rationales that should guide the maintenance of different kinds of materials. In the case of plotting code and most research code the guiding rationale is preservational, whereby members attempt to keep code around, but they keep it largely as is, unsynthesized and only lightly labeled. These resources are things that they might need to revisit, or that might be taken up again, but they are not likely to be used again soon or very frequently. By contrast some code produced in the exploratory process was maintained under a revisionist rationale, by which the code was moved, rewritten, reformatted, redocumented, and perhaps refactored in order to make it part of a more robust software package. Shankar (2007) points out how activities of standardizing and formalizing make the record conversant in a larger, official record, such as that of published scientific papers. The revisionist rationale is similar in that the software is first selected out of accumulating research

scripts, and then remade into standardized, interoperable forms in order to integrate into a collective software package. There it functions as both resource and as record, codifying at least one ‘correct’ and canonical version of an operation.

Another aspect of this distinction is that for the most part research code was developed in a personal way. Research code was sometimes visible to the rest of the Radio Group during lab meetings when a student would make small changes or double check particular parameters they were using, but it was not typically inspected in detail by others. As described in the last chapter, the act of contributing involved working in public, making contributions that would be reviewed by others, and writing documentation or tutorials that would be used by others. This connects with what Kery and Myers (2017) have pointed out about exploratory programming, that it often is difficult to do collaboratively, in part because of its high ‘viscosity.’ Viscosity in this case refers to how difficult it is to make small changes to a codebase (Green and Petre, 1996). This becomes more difficult, for instance, in non-modularized codebases where changing one part of the codebase requires altering many others as well.

This distinction between preservational and revisionist rationales is a kind of bifurcation, in which resources in the lab either undergo an extreme amount of synthesis or almost none at all. This bifurcation is one aspect of the distinction between research code and collaboration code made in the last chapter. In the next section I will look in particular at the ‘selection’ activity, the point of divergence between research code and collaboration code.

6.2 Graduating Code

The process of graduating code is a point of transition between exploration and production, and it is the primary way that exploration comes to shape and influence software production. Put simply, graduating code is a process of taking a particular software-based technique that has

emerged in a process of exploratory programming, and turning it into a "robust" or "validated" piece of collaboration code. This is in some ways a fairly prosaic activity of redesigning or refactoring the code itself. However, it is an important process to examine for a couple reasons. It is the point of transition between epistemic objects and “technological objects” (Rheinberger, 1992). In this sense it is a process by which technological actors-under-examination become the black boxed resources which undergird future work (Latour, 1987). It is also a point of articulation work (Strauss, 1988) and particularly metawork (Gerson, 2006) in connecting exploration and production work. Here I will examine this transition point as a site of work and a point of tension in the research software system.

Researchers often had hunches or vague plans about what kind of useful tool might come out of their exploratory work (such as some new RFI detection technique) during the process of exploration, but for the most part graduating code was something that looks back on code written in the exploratory process and sought to refactor it or rewrite it to be a more robust tool. When one was embroiled in the research process, it was often not clear in specific terms what would emerge from the process as a more useful or more permanent piece of code that might need to be reused. For this reason, when researchers did perform graduation, they were often returning to code that had already been produced in an exploratory mode. Lucas, a professor in the CDA described this situation:

> "I think the model 10 years ago, five years ago, was a graduate student comes up with a code that, on one particular case that they've studied, seems to do something good, they publish a paper about it, and then they move on. And the hard part is getting them- we want graduate students to publish papers and get results out and do science. But making sure we keep them around for an extra month or so, before moving on to a new project to say, ‘okay, you've got this code, it's exciting, let's make it production level code that can integrate with everybody else's pipeline when everybody else can try this out again,’ in that sort of controlled way, and exploring whether it's something that they really want to make a default part of our processing" (Lucas, Professor).

Lucas here marks a point after the release of a publication, where the student returns to code that has been produced in order to turn it into a more sustainable resource. The notion of a "default part of our processing" captures well the way that things learned in the exploratory process can become more reliable components of new investigations.

Lucas' statement highlights another aspect of graduation, which is that it was often framed as a matter of discipline. Prior work has highlighted the fact that applying practices of software engineering retroactively is much more difficult than doing it from the beginning (Heaton and Carver, 2015; Basili et al., 2008), and this is in some sense the obstacle that researchers must navigate in the graduation process. The tendency of research work would be to leave code produced in the exploratory process unchanged once some scientific aim has been accomplished, but the researcher needs to make time and cajole themselves into returning to the code. This frames the work of graduating code (going through code review, writing tests and documentation, etc.) as being added or extra work (Trainer et al., 2015). It is necessary but secondary to a primary activity of science.

In this way the post hoc approach to graduation balanced an ethic towards disciplining software practice with an understanding of the necessity of flexible development work. Members of the collaboration would sometimes gripe about others who they felt were writing "hacky" code. When this came up during one of the Radio Group's lab meetings, Magnus associated with the writing of hacky code, saying that he himself had written a lot of hacky code in the past and he understood the need to avoid too much overhead on exploratory development work. However, he added that you have to have "that checklist" of things that you return to later on. In this way the notion of graduation was a particular balancing or articulation of research programming and collaboration code development. Collaboration code development followed research programming sequentially in time, and research activities remained sequestered from the overheads of software production. There were some caveats to this. For instance, during a

different discussion in a lab meeting Mila pointed out that there are certain things you can do even in research work that can make it easier to improve the code later on. She presented this as an exception to the larger dynamic, however.

The process of graduation was not a natural, automatically occurring phenomenon, but rather a kind metawork (Gerson, 2006) had to be actively pursued. In particular researchers needed to step back from the process of exploration and reflect on when and where they should stop exploration and move novel capacities into a context of production. When, and to what degree, code needed to move towards being collaboration code was not always obvious, and navigating that transition was a developed expertise. For instance, graduating code involves a process of selection (see also Shankar, 2007). This involves identifying parts of the code that are produced in the course of exploration that are good candidates to be productionized. This was a discretionary activity that involved a great deal of judgement on the part of the researchers about what code contributed a good candidate. The criteria for this are discussed in Chapter 5, but they typically include code that many others will want to reuse, or code that will need to be reused by at least one person many times. The code and its purpose also had to be relatively well understood. In other words it had to be at the transition point between successive framings of a problem and the emergence of requirements. In this sense selection often occurred at the end of an exploratory process, or rather it helped define an endpoint for at least some aspect of exploratory work.

The skillset and familiarity needed to do this metawork often came with experience in working with a research software system (and not just experience with software production). Within the Radio Group, Mila often performed this role of navigating when software was fine remaining in a hacky condition versus when it needed to have further software practices applied. For instance, one student was working with software that had been written by a prior member of the lab to perform a new calibration technique, and the group discussed during a lab meeting the

prospects for improving the code. They had created a repository for the code, to, as Mila put it, "collect things together", and she suggested to the student that he could look at changing some of the function names, as that was a relatively apparent problem that could be fixed more easily. However, Mila said that "there might be a time where there is a little more structure... but that time is not now." Magnus agreed that at some point in the future they would need to write it in "an incredibly defensive way", but he made it clear that this, meaning the present instantiation of the code, was "research code." The student's work with the code was still in a regime of research, and although some changes, such as creating a repository and changing terminology used in the code, were worth making now, more thorough productionization of the software was not worth the labor at this point. Magnus pointed out the prematurity of trying to apply these other techniques at this point, saying "if you design it now you will have to design it again." These negotiations render graduation as a subtle process, which is dependent on estimations of the code's broader potential usefulness, but also on self-awareness about the stage of work that one is in, and when certain changes to the software are worth making.

Points of graduation were also transition points between ways of working, and therefore could be sources of tension between the practices of research and software production. Mila for instance described a situation where she had pushed on graduating code and received some pushback. The situation had occurred on a conference call with other members of the CDA, where they were all working on some code in computational notebooks:

> "So originally, when they wrote the notebooks it was like huge code blocks, and then the plot, and then huge code blocks. And I'm like, god, you gotta get these code blocks out of the notebooks. You got to put it somewhere else, and then call them from the notebooks. [...] Well, it's no longer just plotting code. There's like actual complicated analysis including taking delay spectra and massaging them and doing cleaning on them and all this other stuff, [...] As we dug deeper and deeper, it was like, nobody quite knew exactly what the code did. It wasn't really well documented. It was all of these bad problems that happen when people write code quickly and don't think about reusing it,

> and it was now getting used for something other than what I was doing originally" (Mila, Research Scientist).

Mila describes and unease with the way the code has grown in the context of the notebook. Although there has not yet been any explicit error or breakdown, Mila senses that the code has transgressed the boundaries of the research programming environment. This is in part due to the fact that it is performing sophisticated analysis but the developers are unsure of exactly what it is doing, and in part because it is being used for a new purpose without having undergone the process of becoming collaboration code. Mila described pushing pretty hard on this situation:

> "And I came down like a ton of bricks. And then I backed off, because Harvey pushed back pretty hard on me. He said, ‘Look, this is just commissioning data. This isn't actual-really processing data. This is we're trying to figure that stuff out. This is okay.’ But absolutely acknowledged that as this stuff becomes more mainline it needs to move into other repos and get really validated" (Mila, Research Scientist).

In describing this to me I got the impression that Mila regretted somewhat pushing on her collaborators as hard as she did, but also felt that the issue was an important one, that the code was in a risky place. Harvey responds to this with his own contention for the bounds of research code and collaboration code, admitting that it does need to move into a repository to get really "validated", but also arguing that this code is not yet being used for publication analysis or to process "real" data. In this way negotiations over the graduation process were negotiations over where research code ended and where collaboration code began, and the understandings of rigor or validity that undergirded those transition points.

This example demonstrates another aspect of the graduation process, which is that it leverages the differentials established by the repository. What is at stake in this interaction is the movement of code from one established space to another, from notebook to repository. Given the construction of these spaces as being for different kinds of practice, the CDA has practical and relatively clear processes for moving from one kind of practice to another. In order to "really validate" a given piece of code, one moves code from one context to another, and in so doing

one will be required to pursue specific practices for rendering “research code” into “collaboration code.”

6.3 The development of capacities

The research software system is not primarily a system for producing software, but rather it is a system in which developing software is a primary way to develop new modes of encountering and rendering phenomena. Consequently, if we want to understand how a research software system develops over time, we must look at how it produces new modes of engagement over time. This is akin to approaching the accumulation of software in terms of the growth of its functionalities rather than the growth of the lines of code. Looking at the emergence of capacities can help us do this in the context of the research software system. Capacity is a term that has a broader scholarly usage, particularly in actor network approaches, but here I am looking at specific kinds of capacities that researchers can develop by designing and building new instrumentation. In this section I take a retrospective view of the development of the Radio Group’s capacities for RFI detection and its relationship with the cosmo-rfi package. Through this example we observe a couple of characteristics of software instruments and capacities. While this view may be less useful to those focused on the architectures of software components, it is perhaps more useful to the historian or sociologist of science, whose consideration might be on the productivity and development of instruments in research programs.

The notion of capacities takes the phenomenon and instrument together. In doing this, the notion of capacities is similar to a great deal of prior work which has tried to move from reduced or isolated conceptions of artifacts towards consideration of the relations in which they are constituted (Star and Ruhleder, 1994; Jewett and Kling, 1991). This is a crucial shift from software as a generic artifact towards software as embedded in and constituted by broader

trajectories of action. Looking at development in the number of lines or the number of file interdependencies can be valuable ways of seeing change in software, but looking at capacities allows us to see software in relation to a developing research program. In Figure 12, for instance, we can see how the research projects of different graduate students produce new techniques for identifying and removing RFI and how these new capacities can be incorporated and maintained over time. While this is certainly not a thorough historical ontology of RFI phenomena, it does capture in broad strokes a process of technoscientific change (Ribes and Polk, 2014) in the objects and techniques of RFI detection.

However, the notion of capacity is useful also because it adds a consideration of the instrument to the discussion of scientific phenomena. Capacities are also not just phenomena or the "galilean objects" (De Boer, 2021) of science, divorced from the instruments through which they are known. This is visible if we look at some of Mason's early work on RFI. Through his work he established that particular features in the data were digital television (DTV broadcasts) that were reflecting off airplanes moving along the horizon from the array's perspective. If we took the capacity to be only the thing produced, an object made separate from the conditions of its production, then it would be very hard to understand what has been gained by the detection of an airplane. However, the airplane-as-seen-by-the-interferometer, the interferometric airplane, is a new production. The capacity that has been developed is a novel thing. Looking at this form of novelty, rather than only at entirely new entities, allows us to see change in ways of engaging and interpreting the world, and not just change in the major ontological categories of science. In this sense, my usage of the word capacity is trying to make good on a historical ontological commitment towards the collapse of epistemological and ontological concerns (Daston, 2014). In other words, capacities capture ways of knowing and things known as a package.

The development of a capacity is more like a refinement than an addition. Phenomenological accounts have argued that technologies mediate human engagement with the world by

“opening and closing possibilities of experience and interpretation” (pg. 34), which is the amplification-reduction movement of technological mediation (Ihde, 1979). I agree with this position in a broad way, in that the development of new capacities becomes a more *specific* way of engaging the world. The attempt to detect the 21cm signal is perhaps the ideal example of this. Across the contributions in Figure 13, the Radio Group’s instrumentation becomes able to identify and then remove more and more kinds of RFI, from digital television broadcasts to packet loss to evil cows. Stars, too, must be modelled and removed from the data. The research projects of students center on learning to identify and engage entities such as airplanes, but the capacities that they build into their software are oriented around removing these features and sharpening their detection of a potential something else. Capacities are in this sense directed or intentional, rather than the gaining of a more complete set of capabilities.

That said, capacities developed around one general purpose can always become productive in new ways when turned towards other purposes. This is one of the benefits of tracking new modes of engagement rather than just their outcomes, the concepts of an intellectual history of science. This is captured in the somewhat different notion of the “dual use” of technology, by which technologies for war could be repurposed into instruments for radio astronomy (Baneke, 2023). More germane to this study, this is a characteristic of the experimental system (Rheinberger, 1992), that it remains open. Capacities that emerge from the research process can feed back into it in the production of yet new interactional possibilities.

The cosmo-rfi software is both a resource for and outcome of these processes of working out new capacities. As Star and Gerson (1987) point out, even anomalies must be actively constructed by an extant system. Moreover, new changes to the system make possible the engagement of new anomalies and new research questions. For instance, Mason’s development of the detection algorithm for rapidly changing or moving RFI solidified certain phenomena (airplanes) but it was from that perspective, from that state of the instrument, that

the initial anomaly that would become the evil cow came into view. This highlights the importance of reuse or repurposing not in contexts of proof or justification but in the extension of prior work (see also Feinberg, 2020). This is similar to a notion of knowledge turns (Goble et al., 2011), but it centers particularly on the reuse of instrumentation and not just on the feedback of findings.

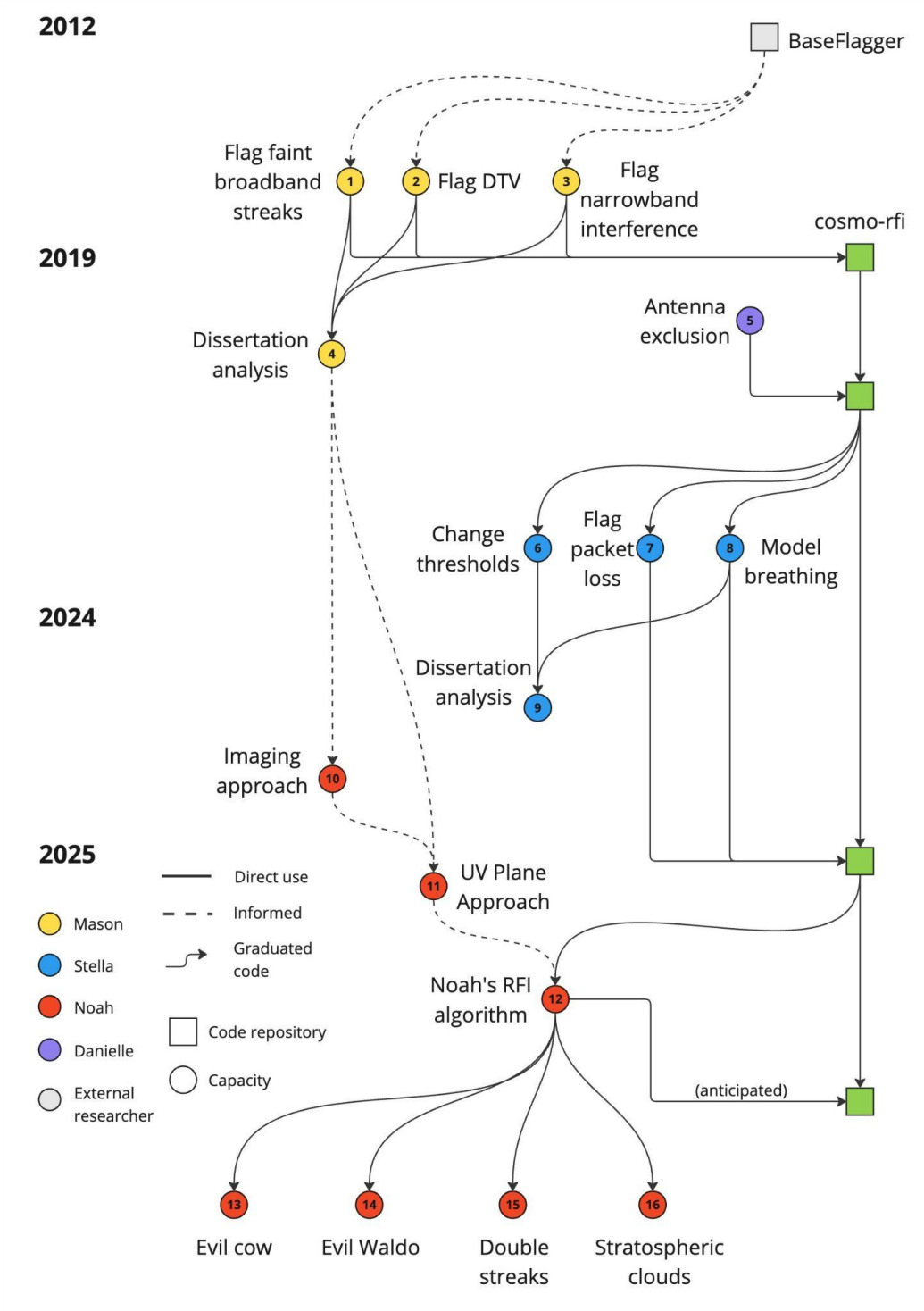

Figure 13: The development of capacities for detecting RFI and their inclusion in the cosmo-rfi software. Outcomes or incidental needs of different research projects in the lab resulted in code being graduated to the cosmo-rfi software, extending its capacities in a piecemeal fashion.

Capacities in my usage are robust and repeatable forms of action, but they are also performative. By this I mean that when the researcher goes to engage a phenomenon or data again (to reproduce or replicate it) they are conducting a new action in a new context, which introduces the possibility of variation or departure from some script that might be defined for this or that capacity. In actor-network terms, the goal of production oriented activity is precisely to create robust networks of actants and to hold them together over time. A capacity is something that researchers understand can be done again with a particular piece of software. It has been accomplished previously and they are operating on the idea that the software will be a reliable resource for similar action in the future. Nevertheless a capacity must be performed again each time, and as described in Chapter 4 each new performance comes with the possibility of variation or breakdown. As with the concept of affordances (Gibson, 1979), capacities are things which must be accomplished each time again.

Figure 13 also illustrates the larger dynamic that robust research software is intended to accomplish in scientific objects over time. Mason's RFI detection strategies were at one point problematic, as were the mysterious objects he was investigating. However, later they return to new research projects as well-ordered capacities, which are ready-to-hand for new investigations. That is, unless they break down for one reason or another given the novelty of the new situation. What was the problematic object of one research program becomes a stable performance for new research work. In this sense the research software system is not just about producing new objects but also about hardening, bolstering, and maintaining networks of actants (Latour, 1987).

A core dynamic for the research software system is accomplishing what Ribes and Polk (2015) call "technoscientific flexibility" (pg. 235), the ability to prepare for ontological change. The development of new capacities requires finding ways of breaking out of present states of working order. However, in order for these things to become capacities, which can be assigned

to a given software package as a quality that it has, they must be re-performable and coherent across different instances. This is a tension that the research software system navigates over time, the intention towards robust, reliable production of phenomena in well-understood ways, and the need to exceed or escape these well understood pathways in order to further the development of the instrument's capacities. For instance, Noah's algorithm for detecting RFI is built in direct comparison with a prior method, and the growth of that capacity relies on Noah being able to re-perform Mason's earlier RFI detection strategies in a robust way, but also to vary a key part of that analysis in order to perform an action that is both similar and different. In the research software system, where production work has been distinguished from research work in process and place, this is also a tension between different ways of working. The balance of durability and novelty in phenomena is accomplished through coordinative means, as described in Chapter 5.

Moving to include the ways of knowing in our account also allows us to examine the long game of scientific change. Critiques have long been made of the act of discovery as a singular momentous act (Woolgar, 1976), and studies of phenomenotechnique highlight that scientific objects emerge "in a long and tedious historical process of purification and ordering" (Rheinberger, 2005, pg. 321). Stories of science, however, are most compelling and seemingly most about 'science' when they involve the emergence of a wholly new object, a quark or a pulsar. None of the epistemic objects I have described here have been disclosed as such entities, a new signal, astronomical body, or rule about the operations of the universe. Looking at these engagements, however, allows us to view the mundane side of epistemic work, the successive efforts to break out of prior states of working order to produce novelty in one's interactions with the world. Any view of the cosmic dawn that the field might eventually produce will come only as the culmination of decades of this development of capacities.

Chapter 7:
Reorganizing rigor in software practice

Part of my goal is to characterize how the Radio Group and the CDA integrated software engineering practices into their research work. This is addressed in part through the distinction of regimes of research code and collaboration code that I described in the last section. That scheme describes how software engineering comes to take a place within the organization of work in the CDA. However, it does not describe how this uptake of practices was accomplished by the actors involved. The typical understanding of how this might happen in the literature is a kind of adoption model, in which researchers take up an explicit set of practices on top of or in addition to their existing practices of research work (e.g. Wilson et al., 2014; Wilson et al., 2017). This uptake is obstructed by misalignment in incentives and reward structures in the sciences, as well as by the problem of unplannability, as discussed earlier.

Here I want to complicate the notion of adoption by pointing out a couple aspects of how the Radio Group went about integrating software engineering practices into their work. First, while my findings certainly bolster the idea that there are conflicts between software engineering work and the reward structures of the sciences, I also point out places where software work becomes a matter of *rigor,* which brings software engineering under a category of scientific work. In section 7.1 I outline some of the complex interactions between understandings of pride in software and understandings of software work as a matter of discipline. These understandings represent shifting orientations towards software work (see Howison and Herbsleb, 2011), but also shifting understandings of scientific rigor.

Second I argue that the uptake of software engineering practices is not a matter of personal, dyadic decisions where an individual researcher encounters and evaluates a given practice. Rather there is a sociality to this process, in which practices are shared, demonstrated, and

pushed through multiway interactions between researchers. In section 7.2 I examine an interpersonal process of *championing*, by which this happens, and then look at the role of repositories as collections of artifacts used to push and define certain kinds of practices in a larger collaboration. Last, in section 7.3 I explore the agency of collections of artifacts–the repository and its interconnected development tools–for shaping and spreading development practice across a larger scientific community.

7.1 Pride and self-discipline in software practice

Prior work has highlighted the disjuncture between reward systems in the sciences and the work of software engineering, as well as the variety of stances that researchers might have vis-a-vis that work (Howison and Herbsleb, 2011). In the Radio Group's efforts to change their practices around software, software work came to be framed in terms of rigor, where software quality, as evaluated in terms of the application of specific software engineering practices, came to be understood as a scientific or epistemic virtue. This was not a unilateral framing, as people often understood that work as being a matter of discipline, warranted only in specific circumstances. Nevertheless, this was a shift in researchers' own understandings of the ancillarity of software development work. Moreover, it constitutes a shift in understandings of rigor, in what constitutes good scientific work. In this section I outline these understandings of discipline, pride and rigor, as well as their interaction with repositories as spaces for public contributions of code within the collaboration.

The Radio Group was part of a larger, active effort to change the way they build software. After one paper in their larger collaboration had to be amended after publication, there was an effort to change various aspects of the collaboration's research process. While this publication error was mostly due to a mathematical issue, software was wrapped up in this larger movement. As Harvey explained, there was an agreement at the collaboration level to write "good quality

software," and this resulted in rewriting some of the collaboration's primary software packages "from scratch." It also involved instituting specific standards for software packages that were to be run on data used in a publication. This included documentation, unit test coverage above 95%, code formatted to a standard, and processes of code review.

I approach this transition as a matter of *rigor* because it approaches the understanding of quality in scientific work as a matter of sensitization to the materialities and procedures of software development work. As described in Chapter 5, the methods that researchers used to understand the quality of software could not be personal knowledge of the code at a low level. Impressions of quality came from knowledge of how the software was developed, as well as indicators like the presence of unit tests or the number of other people who had used it successfully before. In some cases this was a matter of sensitivity to risk in the development of software (see also Kelly, 2015). Harvey, for instance, described this feeling:

> "Well, I think a lot of us have a sense of, I don't know, incompleteness. You know, they say when you see something safe, you know it or not. That's not safe. You kind of know it in your gut, right? [...] So I think a lot of us had that sense, and I think some of us more than others because we'd been burned by various problems maybe more than others" (Harvey, Professor).

The term rigor as I use it here is meant to capture this "gut" feeling aspect of scientific work. It is a way of accessing researchers' own understandings of what constitutes good scientific work for a view of science as practice (Pickering, 1992), rather than for an intellectual or conceptual history of science. Rigor is about learned experience with the peculiarities and behaviors of an instrumental system. In the Radio Group's context this meant sensitization to risk in the material practice of software development. We can see this, for instance, in Mila and Harvey's tensions over graduating a particular piece of code (discussed in section 6.2), where the urgency for graduating the code turned on the organization of the code in a notebook and the fact that nobody could remember what code was actually running. In these circumstances, nothing

explicit has yet gone wrong, but members of the CDA had a sensitivity to conditions that are conducive to silent bugs or other time consuming software issues. This is both in the long-term possibility of having a retraction, but also in the day-to-day work of trying to make sense of tenaciously incomprehensible anomalies.

To be clear, all of the researchers I encountered in the field were long-time programmers and were not developing these sensitivities due to a new uptake of programming in general. Rather, old understandings of dangers and breakdowns in software systems came to be reframed in relation to practices for developing “good quality software.” As Kongsvik et al. (2020) describe in the context of seamanship, the concept develops in its content in response to the movement of new actors, in their case specific seafaring technologies and proceduralization. In a similar way, good scientific software development took on new associations and performances in the Radio Group and CDA.

In particular, rigor in software practice was often discussed as a matter of discipline in the sense that they were understood as something that ought to be done, but which were hard to motivate oneself to do. Noah, for instance, described how most of his code was sitting in a "working" directory and he wanted to contribute some parts of it to the cosmo-rfi software repository, but that was something that "I kind of keep putting off." He said this to me as he was showing me one of his scripts where he had been working on a particular research problem, as if to explain that he knew he really should have contributed it already, but it was something that he simply had not gotten around to. This would have been an instance of graduation, involving all of the overhead of writing unit tests and tutorials and so on. The moral imperative towards this kind of infrastructuralization of software was placed in opposition to the demands of ongoing work. In a similar way, Lucas described how software engineering practices were often seen as a "nuisance," but it potentially prevented disastrous outcomes in the long-run.

This latter sentiment highlights a sense of good software development being a matter that comes to bear on the quality of software that is produced in the collaboration. Lucas discussed “doing things right” with regard to software development practice:

> “I think the way the [CDA] analysis has gone has shown that it's a slow process and takes care. And I think everybody's on board with that now, but it felt like there was a time where there really was a drum beating that doing something right was the enemy of getting things out quickly. And so, you know, we wanted- [pycosmo] was about doing things right” (Lucas, Professor).

Here Lucas avoids a distinction between moving quickly and moving slowly, but rather asserts development practice as a matter of rigor, of "doing things right," which is not necessarily in tension with moving quickly. This was not a universally held ethic, as Lucas himself asserts, and in my observations the deployment of software engineering methods as being unnecessary overhead or necessary kind of rigor that saves time in the long run was variable. Indeed, the notion of graduation described in Chapter 6 is precisely the navigation of these different framings, and in some cases they played out between researchers.

The notion of software engineering work as a matter of rigor also came with understandings of pride in having developed robust and reliable software tools. Abe, for instance, expressed pride in the work that he and others had done on pycosmo:

> "It's high quality code that we build and really is just a way that you know, we can make sure, that we're doing good science and we have traceability and repeatability and all these sorts of things. So, I think it's-I'm honestly pretty proud of [pycosmo]. I think we've done a pretty good job putting it together. I think it is a good model for other repositories to follow. So I myself, when I write other repos, I try to make sure it has good doc strings, good test coverage, things like that. And I think it's just kind of the model that we as a collaboration should be shooting for. So I think it's pretty good...” (Abe, Research Scientist).

Here again, Abe connects good software engineering practice to “doing good science.” This is significant in this case because pycosmo was considered by many to be “glue code,” software that enables conversion between formats, connecting pipelines, and generally wrangling data.

However, in working with an instrumental system constituted by many distinct, interoperating pieces of software, Abe takes an inclusive view of the kind of work and the kinds of software that make good science. It is not just the new algorithm developed by a researcher working on a particular problem, but also the many transformations and interoperations that need to happen to bring the data to the place where some new algorithm can be properly applied to it. Within this scoping, the careful engineering of the pycosmo software falls within the remit of science and constitutes a matter of pride as good scientific work. The development of collaboration code was seen as a matter of discipline but also of rigor, of building tools that would be sound rather than "hacky."

There were also sentiments of obligation towards software engineering practice that were connected with collaboration code as a public and collaborative endeavor. A number of members of the CDA, particularly Ph.D. students, discussed their software work with a self-awareness about it potentially being critiqued. For instance, Stella, a Ph.D. student in the Radio Group, described making what she felt were compromises in a contribution she made to pycosmo:

> "Okay. And even still, like I'm thinking there are parts of the code that I feel like are kind of clunky that I don't really like. Especially after it's been- I think it merged in December. So it's been a couple of months, and I'm looking at it. I'm like, 'oh, boy, that's kind of rough.' But for me at least I hit a point where I was like, I just want this thing to work. And to be part of the thing. And then ready for something else. But now I look at it. I'm like, 'Oh, I will probably rebuild this whole chunk. And I might rebuild this whole chunk.' There's actually a couple of pretty big chunks in it, that I think maybe could be done better" (Stella, Ph.D. student).

This self-awareness of code quality was particularly around code that was contributed to a repository as collaboration code. As mentioned in Chapter 4, the Radio Group operated around research code as a quasi-private artifact. Lab meetings involved inspections of the output of the code but not often inspection of the code itself. In contributing code to a repository researchers

were moving from this relatively private space to a space where their code was able to be inspected by others.

I felt a similar sense of self-awareness in my own contribution to pycosmo. I worried that the code I had written would be sub-par or that I had introduced some kind of hidden error into the code. On a couple of occasions when errors turned up in development meetings, I felt a brief sense of panic that it might have been introduced in the functions that I had added to the code base. A number of the other Ph.D. students expressed similar feelings of exposure or self-awareness about contributed code. Ellie, a Ph.D. student at another laboratory in the CDA expressed some amazement that Mila would frequently code live during pycosmo development meetings, sharing her screen to work on a problem while others watched. She also felt some embarrassment because she felt behind in developing parts of the code she had been assigned to work on:

> "...whenever anybody mentions [pycalibrator] I will just start cringing because it's like, oh, I'm supposed to be doing [pycalibrator]. Oh, no, this is my fault. Of course it isn't, but..." (Ellie, Ph.D. Student).

It is worth surfacing these sentiments because they capture the way that collaboration code entails understandings of responsibility. The act of contributing was to make one's work visible, and in some sense to make it liable for the quality of analysis done with the code.

Kery and Myers (2017) argued that it is difficult to do exploratory programming collectively and I have noted a similar distinction between research code and collaboration code. Research code is often written by and for a specific person, and it is personally maintained and organized. Students would bring research code to meetings, and it was occasionally inspected (rather than the plots it produced, which was more frequent), but they were generally left to their own devices to organize and maintain that code how they saw fit. Collaboration code on the other hand constitutes a more public space. It is not only *for* a broader group of people, it is also

visible to a broader group and it undergoes review by other researchers. There were a couple of instances of software in the group's larger field that had been written by a single individual and that were largely maintained by that person, but these were uncommon, and the term "collaboration" in collaboration code typically implied a collectively developed tool.

Understanding these associations with software engineering practice are critical to understanding how it comes to be justified and mobilized within the research group and collaboration. In some cases software engineering work came to be categorized as a kind of work in tension with science in the sense that it obstructed getting things done quickly. However, in other contexts it was framed as within the work of science as a matter of rigor, as well as an investment of labor that would save time in the long run. The shifting between software work as external or internal to the scientific concern was a kind of boundary work (Gieryn, 1983) that sought to define what aspects of infrastructural work were in or out of a primary project of science, with ramifications for justifying and validating that work. Nevertheless, through activities of championing and the development of exemplar repositories, members of the CDA do posit a model of scientific rigor in which moving slowly, taking care, and following procedure are epistemic virtues.

Overall it did not quite accomplish the holistic transition in understandings that Kongsvik et al (2020) describe, where seafarers shifted their understandings of personal competence to a kind of capability distributed through various technologies that had entered their workplace. As they argue that case involved a fundamental shift in understandings of role and competence. The Radio Group and CDA did not see fundamental shifts in their understanding of competence or rigor, and there remained some vacillation and contestation over understandings of how 'good scientific work' related to software development. Nevertheless, where software engineering work was incorporated into an understanding of scientific work it entailed pride, obligation, and discipline towards software work despite the fact that researchers were not working in a

software subfield where software products were their primary output. While this definition of software work as germane to scientific concerns is not a solution to the underfunding and recognition of software development work amongst scientists it is an important aspect of shifting motivations and justifications of that work.

7.2 Championing

In discussions about the adoption of software engineering practices, often rationales for the adoption of such practices are provided (sustainability, reproducibility, and so on) but a lot is left to the imagination about how such adoption might actually happen within a larger scientific community. This leaves one to imagine that researchers will encounter these practices in principle, see their potential value, and then go figure out how to implement them. There are some discussions of adoption at a community level (Koehler, 2020), but still does not get us much further towards understanding how such practices get taken up and spread amongst a community. In the Radio Group and amongst its collaborators, the uptake of software engineering practices was a complex matter, involving social processes of sharing and demonstrating development practices and tools, as well as certain people taking on distinct roles in promoting software development. In other words there are a number of complex dynamics to how a research collaboration takes up practices, understands how to do them, understands their emergent usefulness to scientific work, and begins to actually commit to the work of implementing it on a regular basis.

One central aspect of these dynamics was *championing,* a process by which certain members of the collaboration took on a role of promoting software development techniques. This could be as simple as advocating for paid time for software work from grants funds, a point which was ascribed to certain PIs. More often it was a matter of calling for or advocating for the use of

particular software engineering practices in certain contexts of development, such as in when to graduate code.

Championing also involved learning new practices or techniques for software development and bringing them into the collaboration. Abe pointed this out about Mila's involvement in pycosmo:

> "I mean, again, I think the main driver has been pycosmo. Mila leading the charge, learning something new, seeing how it could be useful. And, you know, I think the response of a lot of people would have been well, that's just seems like a nuisance, and-putting it in anyway, and then a year later saying, 'Wow, that just saved our butt, and prevented us from making a big mistake by catching this bug that somebody introduced.' Once they prove their value, there's other adopters" (Lucas, Professor).

Part of Lucas's characterization is, again, a matter of discipline, a matter of doing work despite seeing it as a nuisance. However, another critical part of this is a process of learning and implementing. He references Mila learning external practices, but also figuring out how to implement them in local sites of work. Mila's time was in fact split between the Radio Group and an organization called the Data Science Institute, which was focused on developing computational methods in the sciences. In explaining the development practice on pycosmo, she ascribed it in part to her interactions there:

> "And both Lucas and Harvey were aware that there were good practices in Python around testing and quality of code that weren't necessarily being followed in [CDA]. And they didn't necessarily know all of them but knew some of them. And I just moved up to [the Data Science Institute], and I was becoming aware of some of these things. [...] And in that process we rewrote the code that they'd written over the summer to make it much more robust. And I learned Python, and how to write tested Python code in that process" (Mila, research scientist)

Mila ascribes the work of Lucas, Harvey, and herself in part to an awareness of external practices, and in her own case the process of learning testing practices on an external project. This aspect of championing is more like a process of organizational learning or boundary spanning, by which certain individuals adapt practices from elsewhere into local contexts of work, where they might be more easily picked up by others.

An important aspect of this process of sharing practices was a process of evaluating and weighing the value of tools that could potentially be picked up. In one lab meeting Mila told the group about a talk she had seen on using a Git-like versioning system for data. Aware of her own role as somebody who often brought these kinds of tools back to work in the group, she indicated a number of times that she was initially skeptical of its usefulness, but that she could see some value after the talk. In discussing this with Magnus they seemed to agree that the potential value of such a tool would be in its “high level” functionality, which would make dealing with the nitty gritty of versioning easier. In the end they simply decided to keep an eye on the tool in case a more serious need for it came up. This kind of haggling over what a tool specifically does and what it might be useful for was a critical part of what a champion did, but it had little to do with implementing or writing code itself.

I have termed these kinds of things championing to indicate that they are activities that are done in certain circumstances, and I saw a number of people amongst the Radio Group’s collaborators perform championing in a number of different contexts, but championing was also something that was associated with particular people, who were frequent and vocal proponents of software practice. 4 or 5 other members of the CDA referenced Mila as one who frequently promoted new development practices. Harvey indicated that it was Mila, in part, who "has us doing this good stuff", and Danielle referenced Mila as one reason why pycosmo was developed the way that it was:

> "I think it's probably partly just the people who wrote them. You know, Mila has very good coding practices and I think has pushed really hard on pycosmo to be a useful and well-documented tool" (Danielle, Ph.D. student)

A number of other names were mentioned as well in explaining the general uptake of software engineering practices in the larger collaboration, but it was something that became a persistent role in the community.

The most active champions of software practice were research scientists. Howison and Herbsleb (2011) observed differences in the stances that different research groups have towards software, such as their working in a software-focused subfield or as a byproduct of research work. The members of the CDA were almost all researchers first and foremost, meaning that scientific contributions were their primary object of work. However, smaller variations of this existed in the Radio Group and the CDA in terms of seniority. As noted previously the development meetings for pycosmo were dominated primarily by research scientists and some professors, with Ph.D. students attending occasionally. Almost all students in the Radio Group contributed at some point to a repository and went through the processes of code review, writing unit tests and documentation and so on. However, the more senior members of the group typically directed them towards research work and the investigation of anomalies, while making occasional contributions to more robust software packages. My discussions with Ph.D. students had few of the firmly-held normative stances about software quality that discussions with some professors and especially research scientists did. It was also in many cases research scientists who brought new development techniques into these projects. As stated above many of the developers working on things like pycosmo were research scientists and their week-to-week development of the software involved finding new tools to support it and making conscious decisions about development processes on the project.

Championing could also be a source of interpersonal tension where the promotion of best practices became a critique of existing practice. In discussions of championing in the literature, there is some acknowledgement of resistance to a change, but the depiction tends towards rosiness in characterizing champions as savvy, resourceful individuals who bring about positive, generally welcomed change (Beath, 1991). Championing, however, involved contestations over when to graduate code, for instance, as described in Chapter 6. In discussing these dynamics in an interview, Mila seemed to regret having come down "like a ton of bricks", but also defended

her conviction about the importance of the issue. When I discussed software development practice with Harvey, he mentioned the same incident:

> "At some point during last observing season I was showing more plots from the nightly notebooks, which we were talking about before. And Mila says, ‘wait a second, another question, where's this code? What repository is this code in? It's running outside, it's running onsite and it's not in a repository?’ And just kind of that's- the conversation came about. It's one of those things where, as a group, if we don't point something out right away, sometimes it just falls away" (Harvey, professor).

Harvey situated this incident in a larger dynamic in the collaboration where different members will sometimes "hassle" each other over the details of their software practice and the potential risks. Moreover, these processes are a kind of collective discipline, by which the collaboration as a group can stay on top of the problem of maintaining software quality. Practices which are not raised and made visible this way, by a process of championing, can simply "fall away." What is cultivated here is a kind of attentiveness or discipline, and it is a group effort and not only an individual one. Harvey described this at a higher level:

> "I mean, we have a kind of community marriage, you know, you take turns being the bad guy. Whoever happens to be most sort of, most recently burned by the particular problem pointed out… so sometimes it's test coverage, sometimes it's tutorials, sometimes it's, you know, community stuff" (Harvey, professor)

Championing here implies a kind of agonism, where certain parties are called out for their practices, and, simultaneously, the champion takes on the role of the "bad guy", one who is critiquing existing ways of working.

Championing was also often understood as a laborious kind of work, in part because of the interpersonal tensions that accompanied it and in part because it often implied overhead development work. In describing champions and championing, researchers’ language tended towards strenuous effort, describing them “pushing” for better software practices, people

“hassling” each other, or “fighting” for certain ways of working. Mila referenced this dynamic in reflecting on change over time in the collaboration:

> "...I felt like I was sometimes the one fighting that fight. And now it's like, I don't even have to push that hard because other people jump in and say 'no, no, that doesn't make any sense,' because they've all been involved" (Mila, research scientist).

"Pushing" was a term often used to describe activities of championing, capturing a sense of resistance. It is captured here also in the notion of "fighting that fight." Mila however also references change in championing-as-tension as more people engage in the promotion of new development practices. This reflects a change in the collaboration’s collective understanding of good software practice.

Overall, championing was a dramaturgical[6] thing, in which certain people took on certain roles and reputations as promoters of software engineering techniques. This role was in part one of authority, of being able to influence the way software was built in a larger collaboration and make calls about the usefulness of different tools. It was also, however, a process that sometimes involved conflict and pushback. This is a critical dynamic both for understanding exactly how it is that software engineering practices spread through a collaboration or sub field, but also in the sense that these kinds of interpersonal tensions are something that any collaboration hoping to change its practices around software will need to work through. Navigating such tensions exist over and above the ‘extra work’ of software engineering practices (Trainer et al., 2015).

[6] By “dramaturgical” here I mean the way that I came to code these issues as dramaturgical codes (Saldaña, 2021). By using the term championing I moved away from individuals’ decisions to adopt or not and towards performances, roles, and interpersonal interactions around the process of collective adoption. I do not mean to leverage the larger machinery of dramaturgical sociology, but I would argue that it does shift our perspective on what adoption is in a productive way.

7.3 Champions and exemplars, human and nonhuman

Thus far the championing I have been describing has been about people taking on a particular role within a collaboration, but nonhuman artifacts had significant kinds of agency in the pushing and promotion of software practices. First, as described in Chapter 5, constellations of artifacts embedded in repositories could enforce certain kinds of practices around the code, such as writing tutorial and documentation or aligning your contributed code with other parts of the software or other software packages. There are some additional ways that software repositories performed agency, however. A second way was in serving as an exemplar, an accessible and well-documented example of how to use particular kinds of practices, what tools to use, and what 'good' software looks like in a practical sense. As an exemplar, a software repository *demonstrates* good practice. Third, repositories constitute a place for a particular kind of work, and in that role they provide the context for discussion, negotiation, and teaching of practices. In all of these ways it is possible to discuss the repository as a *vector* for software practice.

The situation of a software repository being an exemplar was perhaps the most direct way that artifacts played into the promotion of new practices. Harvey particularly referenced Pycosmo as a software repository that served as a "prototype" for the implementation of new development practices:

> "So we had to start from ground zero. And so we got- recognizing that and I think it [Mila] went and figured out like how do you do good software engineering and came back and told us, and then we did it for pycosmo. And that has sort of been a prototype for everything else that we've done everywhere else" (Harvey, professor).

Harvey here is describing a kind of sharing of software practice, but it happens through a particular software package standing as an exemplar of how to do good software practice. Importantly, this is not just a process of naming or identifying the proper practices. It is a process of *demonstration,* in which the practice or tool is shown implemented, indicating not

only what the thing is (such as “continuous integration”), but how to implement it in a practical sense and why it is useful. Mila described this aspect of pycosmo as well:

> “We adopt things like continuous integration.People see it's useful and go steal our CI things and then tweak them to make it work for theirs. We adopt things like pytest as opposed to nosetest. We figure out how to do it- what the changes have to be, what they look like, people start seeing it there and they go, ‘Oh, we should be using pytest’ and then they move their repos to pytest. We adopt Sphinx documentation. And that's been less- some people have done that, some people not so much, but…. Yeah, people see things in [pycosmo] and they're like, ‘Oh, that's nicer than what I'm doing. I should do that.’" (Mila, research scientist)

The presence of these documentation and continuous integration tools on pycosmo first suggests specific tools to use for a given practice, such as continuous integration, and suggests one that is likely to work and be manageable for similar kinds of software packages and developers. More than that, however, it shows the tool in its context, in terms of the specific configuration you have to set up in order to make it work.

As discussed above, researchers sometimes took pride in the quality of the code that they produced. In this sense code stood for people and their work, and could itself be a statement about the importance and the particulars of “doing things right” as Lucas put it. Abe, one of the developers of pycosmo, described how the software could “stand for” good software practice:

> "Because, you know, it was also kind of a way for us to say, 'what do we want our repos to stand for?' and I think it was good software practices as defined by, you know, good documentation, high unit test coverage, high quality code that has been kind of rigorously checked, and has had a lot of eyeballs on it" (Abe, research scientist).

Abe also put this in terms of embodying particular ways of working:

> "We, as the [developers], kind of see pycosmo as our flagship repo. We try to practice what we preach and have best software practices, you know, good docstrings, test coverage as much as we can write, continuous integration, those sorts of things. So I think it just like in the sense that we want to, as best we can, embody what we see as good software practice" (Abe, research scientist).

As collaboration code, the development of pycosmo is highly visible, and contributors sense external evaluation of the code, whether in terms of potential embarrassment or in terms of pride. For researchers who have a strong commitment towards development practices, then, the development of the software becomes a venue for the statement and demonstration of those practices.

In embodying notions of "good software practice" a repository like pycosmo becomes an *exemplar*. Lucas referenced his own process of having learned from pycosmo as an example, saying that "for the most part for the astronomers, we've been learning by watching pycosmo" (Lucas, Professor). He further attributed long-term changes in the collaboration at least in part to having the example of pycosmo:

> "I think the software development practices, which again, I think a lot of people have now borrowed or learned from their work on pycosmo within these collaborations. If you look again five years ago, things look pretty different on the software landscape, and it really was Mila putting these things in pycosmo and other people seeing their value that led to their adoption" (Lucas, professor).

Lucas again discusses both Mila and pycosmo as strong proponents of software practice, but he also highlights another aspect of demonstration. As he points out, people not only see things implemented on the repository, they have also learned how to use them and implement them in that context, making it easier for them to do it again elsewhere. This happens in particular in the practical activities of contributing to the repository. I, for instance, had never encountered an automated formatter, but when contributing to pycosmo I saw exactly how it worked: it ran using a “pre-commit hook” that would automatically reformat my code before it was committed as a change to my working version of the software. This kind of demonstration shows a tool *in situ*, embedded in a context of use. It also enforces its use on one who may not have even known about the tool.

Repositories could also serve as testing grounds for techniques that might be used elsewhere. This dynamic was not only from pycosmo to other repositories. For instance, on one occasion when the pycosmo team was looking at how to implement a new packaging system, Mila first implemented it on another repository with many of the same developers, because that was a less complicated piece of code which could serve as a testing ground. Once she had worked it out there it was applied to the more complicated pycosmo. In this way some repositories could serve as testing grounds or test cases for practices that were as of yet poorly understood.

Repositories could serve as exemplars, but they also served as meeting grounds where practices could be shared between developers. For instance, during one pycosmo development meeting Mila showed other members of the team a parameter she had discovered that would cause the testing process to exit on the first error rather than continuing to run through the rest of the tests. This saved a great deal of time and allowed focusing on just one failure at a time while debugging. She was sharing her screen at the time and showed how it worked on tests that they had just been running. There was a chorus of "neat" and "that's useful" from other members of the development team. In another instance Mila was adding in a warning message in a number of places by copy/pasting the message in the relevant parts of the code. Sam, another research scientist who was on the call watching, said that she could just make a variable for it, rather than using copy/paste so much. Implied in this comment was the goal of avoiding copy/pasting code frequently, an often cited best practice (Wilson et al., 2014). One rationale for this is that copy/pasting would create lots of versions of the message in different places which would all need to be updated separately or removed when changes are made. Mila agreed that creating a variable was a better idea than "copy/pasting all over the place" and switched her strategy. These kinds of interactions are the practical mechanisms by which development practices spread from one researcher to others, or simply how they are promoted.

In my own work on the repository I experienced how it could constitute a material basis for teaching or sharing practices. I was a novice at using Git versioning systems on a collaborative software project, and so I had very little understanding of how to build up commits in a way that would produce an interpretable Git history on the repository. While working on a contribution, I "pulled" the old version of the code into my new version in order to merge it. A few minutes later I got a message from Mila on the group messaging platform:

> Mila:
> Hi Will, it looks like you're merging master into your branch? that may make the final rebase ugly.

Mila had apparently seen the changes I had made to the "branch" of the code I was working on, and had noticed that I had made a move with Git that would cause problems down the road when I had to reintegrate my code into the main branch of the repository. Through the following exchange we figured out what I had done exactly and what the preferred approach:

> Will
> Oh really?
> I thought I did a rebase onto master
>
> Mila:
> Oh! then you probably did, but at the end you did a git pull instead of a git push -f
>
> Will:
> oh, yeah
> I pulled my remote branch
> is that a bad idea?
>
> Mila:
> yeah, the pull merges your remote branch into your rebased branch, which you definitely do not want to do. It gives you multiples of each commit
> It's confusing, because it's the hint git gives you when you try to push after a rebase
>
> Will:
> ah ok, can I undo that somehow?
> yeah I think I read that

Through discussion we identified a specific mistake, that I had rebased my new code on an updated version of the main branch of the repository and then, following a message in Git itself, pulled the remote version of my code into my new rebased code. The main point in this is that it was unnecessary and would create a mess of the Git history by creating duplicate representations of particular commits. This would be problematic particularly when I went to merge this messy history back into the main branch of the repository. Over the next 10 minutes Mila walked me through how to untangle my Git history and then suggested a tool for monitoring Git repositories that would make it easier to see some of these problems.

Interactions like this highlight how the software repository itself provides a materially agentic space for the promotion and teaching of new software practices. As mentioned above repositories like pycosmo required programming in a public way, where others would examine your code (and Git usage). For this reason my mistake with Git became visible within a common space, specifically through a shared Git history that we were all working on. Mila can also see my actions and point to the place where I diverged from desired practice and also explain why it would become a problem. She could then describe a preferred practice, connect it to the problem it was meant to address, and see that I did it correctly. Finally, she then recommended an external tool, which in the context of this mistake I had just made had a clear purpose. It was now clear to me what such a tool would be *for*. Putting all of this together, it is important to recognize how the repository serves as a collection of artifacts *around which* championing and teaching of practices can occur.

Lastly, it is worth considering a couple aspects of a repository like pycosmo that make it a good 'vector' for the spread of development practice throughout a collaboration (and beyond). One is that because pycosmo consists of lower-level format conversion and data manipulation functionality, it has a large user base. Abe made this point in explaining why pycosmo gets more "attention" than other repositories:

> "[Pycosmo] gets most of that attention. And I would make the argument that it's justified because it has the largest user base of any of our software repos. And it has the greatest probability of being seen and picked up by an external group, for instance the SMA. They were interested in [pycosmo]. They weren't as interested or maybe hadn't heard of our simulation package, [software package]. So I think that is so... we want to put our best foot forward with [pycosmo]" (Abe, research scientist)

As Abe points out, pycosmo is a good candidate for implementing these practices because many people rely on it, but in its role as an exemplar it also has a large audience. Closely related to this point is the fact that pycosmo is an interorganizational project. This means that members of the development team on pycosmo share development practices while working on that repository and then return to respective labs and development projects and have the potential to implement them there. While many of the development projects in the CDA are carried out across labs and universities, pycosmo is particularly interorganizational, involving contributors from at least 4 universities, who represent three separate collaborations or telescope projects.

It is important to take seriously the repository, and the project, as agents in a broader change in software practice. Many of the dynamics of championing described above involved learning and bringing new practices into the context of the collaboration. The pycosmo repository in particular created a venue for the initial adaptation of these practices and their demonstration to a larger group of people. It is important to consider this dynamic in understanding the larger process of change in software practice in the CDA.

Chapter 8:
From scientific software to research software systems

The primary goal of this dissertation has been to put forward the notion of the research software system as a way of rethinking software as a tool for scientific work. At the highest level the concept recasts the problems of scientific software in the framing of research infrastructure, in terms of how scientific communities reconcile rigidities or perdurance of infrastructural software systems with change in stakeholders and research topics. In doing this it helps us understand how researchers build and use research tools in the face of unplannability and the absence of requirements.

At a lower level, the concept makes a couple of contributions. Firstly, it asserts the existence of exploratory work as an unpredictable but highly structured activity that is distinct from software production in its function, temporalities, and outcomes. In doing this it highlights that the issue of scientific software is not just an issue of production process. In other words, while the problem of how to get scientific communities to build robust and reliable software is an important concern, there is another, related concern of how software becomes a tool for exploratory, epistemic work. This is a slightly different problematic that becomes apparent when we look at exploratory programming directly.

With the notion of exploratory work characterized to some degree, the concept of the research software system then also points us to the interactions between that process and the process of software production. It points our attention to the fact that graduating code, for instance, is a key point of negotiation and articulation for making the larger research software system go well. This opens up new points of leverage on the problem of scientific software beyond the codification of

best practices. It points out articulation work (Strauss, 1988) as a point of difficulty for the integration of software engineering practices, but also as a place where interventions can be made to make that integration go more smoothly.

In looking at interactions between processes of production and exploration the research software system also extends our understanding of how exactly software engineering practices come to be integrated into scientific work. Prior work has described primarily how such practices benefit sustainability and collaboration. They have also discussed the validity of software to a certain degree, but usually with regard to the specter of journal review or retraction. The research software system helps us develop an understanding of how such practices contribute to the process of developing new understandings of nascent phenomena, an activity usually relegated to the context of discovery in science.

Lastly, the research software system can also begin to problematize how exactly practices of software production might become a part of scientific communities and scientific work. While the concept does not aim to conceptualize this process directly, it can point us to a couple of places where our implicit understanding of how this happens will need to develop with future research.

In this Chapter I will draw out some of the implications of these contributions and attempt to give some scoping to how the sensitizing concepts developed here might become useful in future work. Firstly, in section 8.1 I describe how the software processes described here are specific to the Radio Group, both in their history with software work and the nature of their research programs. In section 8.2 I discuss how the research software system can reframe our understanding of unplannability in science and its relationship to the work of software development. In 8.3 I describe some of the ways that we can problematize the adaptation and integration of software engineering practices into scientific work. In 8.4 I revisit what exactly it is that concerns us with scientific software as an object of study and how the research software

system might make that topic relevant in new ways to both social studies of science as well as to studies of software engineering.

8.1 Software process in and of the Radio Group

While the concepts of exploration and the research software system, as I develop them here, are intended to be useful in a variety of future investigations, the shape the concepts have taken is closely connected with the scientific and organizational context of the Radio Group and the CDA. Here I will walk through a couple of aspects of the Radio Group and their disciplinary and organizational milieu that were salient in the way that they work with software. The goal of this is not to delimit the use of the notion of research software systems, or to place stringent boundary conditions on it. Nor is it intended to be a detailed account of the disciplinary or organizational traditions of the group. Rather the goal is to help the reader track differences with specific contexts where they might be looking at processes of exploration or at research software systems.

First, the larger field of astrophysics that the Radio Group operates in has considered programming (although not software engineering practices) as an essential aspect of their work for a long time. During my observations, the more senior members of the lab would respond to the students' contemporary software problems in Python with their own stories of problems encountered in IDL or in Fortran during their own Ph.D. work. While the software engineering techniques they had taken up in the preceding 10 years or so were certainly novel, this familiarity with programming as an activity certainly provided a strong foundation that researchers in other fields may not have (see also Sutherland et al., 2025).

Another aspect of the Radio Group that shaped their organization of a research software system was the makeup of the field in terms of seniority. Ph.D. students made regular contributions to

the more production-oriented repositories, but the week-to-week work of managing issues, fixing bugs, and doing code review fell largely on the shoulders of research scientists and postdocs, with contributions also from professors. This is a particular delegation of labor that came to work for the Radio Group and their collaborators, and which depended on the presence of a substantial population of postdocs and research scientists.

During the period of my observations the Radio Group also worked within a relatively coherent collaboration context. By this I mean that their group worked with a number of other groups at other universities on a shared instrument and shared software pipelines. There is some distinction between the parts of this analytical system that different groups work on, but there was a shared investment in the hardware of the array as well as certain low-level processing software. Moreover, papers published within the collaboration that presented scientific results often included everyone in the collaboration as an author. A collaboration like this, which is understood to be a shared endeavor to a certain extent, provides a particular context for building shared software systems. While some software, such as Pycosmo, was intended for the broader field, most production-oriented software was intended primarily for a relatively tight-knit group of collaborators and laboratories. Software did not need to jump from being a personal tool to an openly available system, and discussions around changes in software were often aided by the participants already having some interpersonal familiarity with each other (some interactions on Pycosmo being an exception to this).

Much more work is needed to understand how these different aspects of a research group's situation might shape their software practice, but here at least they can serve as contextual information to assist in making use of my findings. Comparative studies or more quantitative analyses might be useful in examining some of these features across research groups.

8.2 Unplannability and the research software system

While this study certainly does not find some magical solution to the problems of working without requirements or oracles, it does help reframe our approach to scientific work with software in a way that is potentially productive. Considering a larger research software system can expand our understanding of how tensions between rigidity and flexibility might be reconciled (or at least managed). As prior work has detailed, software work is often understood as an alternative or ancillary kind of work (Trainer et al., 2015), which positions it as a drag or burden on the ability to get things done or “get plots moving” as one of my interlocutors put it. Additionally, in my own findings the notion of production software has close association with collaborative overheads. By collaborative overheads I mean that researchers must both consult others and their needs in the design of new changes. Spencer (2015) also observes this in describing the bureaucracy of software. In these understandings there is a rigidity to software in the sense of development process, in the work required to contribute and the slowness with which it can change. Moreover, robust or production software development is in opposition or direct tension with flexibility or the rapidity with which researchers can explore new problems.

In line with prior studies of research infrastructures, however, dynamics of flexibility and rigidity are complex outcomes of organizing and coordinative work (Bietz et al., 2012; Ribes and Polk, 2014). Rigidity can also be described in terms of brittleness (Spencer, 2015) or viscosity (Kery and Myers, 2017; Green and Petre, 1996), the difficulty of making changes to a piece of code without breaking things or needing to change a great deal of the rest of code. Under this framing a software package might be considered as flexible precisely because more overhead or extra work has gone into it, given that having documentation, modularity, and unit tests makes it easier (in a particular sense) to change a large, complex software system without introducing errors. These two framings of flexibility versus rigidity are therefore themselves in tension, such

that one can mean flexibility in terms of the ability to quickly try new things, or in terms of the ease of maintaining and extending a larger codebase.

My findings extend these considerations in two ways. First, in Chapter 5 I outlined yet another sense in which flexibility can be understood in research work. As I argued, the development of robust and reliable production code contributes back into the exploratory process in that it prevents artifacts that confound the interpretation of outcome of open-ended tests. That is, because having reliable, tested code assists in the retrospective assessment of what happened and the rapid and clear constitution of new phenomena. As Lucas confessed, he lost months of effort to a bug caused by a mismatch in polarization conventions. When all such conventions and manipulations can be delegated to robust, reliable software components, the researcher has more flexibility in producing and reproducing phenomena without needing to rework and troubleshoot their entire infrastructure. In this sense it is precisely the rigidity of development process (bureaucratic overheads and extra work) which contributes to technoscientific flexibility (change in the techniques and objects of science). In other words the regimentation of production work can give some leverage over the clarity and workability of a scientific community's working objects (Daston and Galison, 2007).

In part this means recognizing production software as a resource for exploration and not only a constraint on it. As D'Adderio (2011) has pointed out we can fall into an assumption that artifacts are constrainers of action, especially when those artifacts are representations of the action itself, such as plans or protocols. Rheinberger's experimental system is once again useful for reframing this idea. The technological objects in his characterization are the conditions into which scientific objects can take shape. They "contain the scientific object in the double sense of the word: they embed it and they restrict it" (Rheinberger, 1992a, pg. 310). The tools that the researcher works with delimit their capacity for new action–the kinds of questions they can ask and the kinds of answers they can get–but it also provides the coherence and the mode of

representation that is needed to establish a robust, recognizable phenomenon, whether it is well-understood or not. An experimental system without regularity in its functioning does not produce coherent phenomena of any kind, whether they be anomalies or well-known features.

In D'Adderio's (2011) terms this means that we cannot account for novelty or generativity in action just by emphasizing human agency against constraining devices. We must look at a more complex interaction by which the resources or repertoires of science might contribute to novelty and change precisely through their regimentation. In my case this creates the somewhat paradoxical observation that the flexibility of exploratory work is maintained in part by the regimentation of production work. Situated assessments of software quality, of unit test coverage or rigor in development process, contribute to the assessment of the outcomes of instrumental tests. They undergird the evaluation of a bump on a plot as a cosmological bump or a bug in the system.

In this way the research software system highlights the organizational or coordinative dimensions of dealing with unplannability, similar to past studies of research infrastructures (Bietz et al., 2012; Ribes, 2014). In section 8.5 I discuss the value of considering software artifacts as instances of other categories (such as instruments), and considering the move towards production software in the sciences as an infrastructuring movement is one way of doing this. One of the key outcomes of this in my analysis is recognizing flexibility as a complex outcome of an organizing process rather than as a simple quality of a particular design choice. To put this a different way, flexibility and rigidity were not associated in a one-to-one fashion with research code and production code respectively. It is not the case that exploratory work was flexible in nature while production work was rigid in nature. As pointed out in sections 5.3 and 6.2, for instance, working with code in an exploratory way for an extended period could result in code that becomes tangled and uninterpretable, without documentation or accessible design. Accomplishing a codebase that was flexible with respect to ongoing and developing

research problems was accomplished through the combination and articulation of processes of production and exploration. This observation is in line with studies of research infrastructure, and provides an account of software as embedded in processes of organizing.

These slightly different but interacting notions of flexibility and rigidity present something of a tangle for both the scientist and the onlooking scholar. The second contribution I make to this set of issues is to present the research software system as a characterization of how a research community itself manages this tangle of tradeoffs between flexibility in development process technoscientific change, and so on. As I have argued, the research software system does not seek to apply the extra work of production software development universally to all software development work, but rather it designates distinct spaces for work with low overheads and for more production-oriented work. If this goes well, it enables researchers to make the most that they can of both software engineering practices and exploratory work. When they need to, they can work without needing to undergo code review, write unit tests, or write documentation. At the same time they can take advantage of robust code as a resource for exploratory work and for the constitution of robust scientific objects. This does not magically obviate the need for extra work in software production, but it mitigates the problems of having rigidity where it is not beneficial while employing it where it is beneficial.

In order to make this management of flexibility clear, we can look at where a research software system fails. In the case described in section 6.2, Lucas and Mila debated whether particular code that had been written in a computation notebook should graduate or not. The code had grown and grown in the course of exploratory work to the point that people had a hard time remembering where different operations were being performed. Mila and Lucas' confrontation on that issue was about whether this situation was more appropriately a matter of exploration or a matter of production. What I want to point out here is what is at stake: if something has gone wrong it is not that rapid exploratory work has happened at all, but rather that it has continued to

a point which could be deemed more appropriate for production work. Mila had sensed the dynamic which I describe above, and which Lucas himself illustrated, that ‘messy’ or ‘hacky’ code could threaten the ability to make sense of new outcomes. Where the research software system breaks down, then, is when practice occurs *out of place*. This is a breakdown in articulation work (Strauss, 1988), in the connection or transition between kinds of tasks.

As a breakdown in articulation work this kind of issue can benefit from explicit attempts to scaffold articulation work (Schmidt and Bannon, 1992). This could be direct organizational policy efforts, such as establishing clear criteria (as clear as possible) for when code should graduate. It could otherwise be reported through tools that support refactoring. This might include, for instance, tools that produce information about code produced in order to help make decisions about when to refactor and what parts of the code can be refactored (e.g. Hayashi et al., 2006). Perhaps the most important observation, however, is that performing the kind of work necessary to effectively transition from exploratory work to production work, for instance, is a matter of developed experience, which is not often codified in sets of best practices proffered to scientists. This experience is something that would certainly be aided by greater experience with software production work, but it is also a kind of experience that needs to develop to some extent in the context of a research software system. As described in section 6.2 the metawork of transitioning out of exploratory modes of work requires familiarity with rhythms of research and publication, the stopping points where a capacity is ‘done’ or ‘good enough’, as well as the assessment of what aspects of a research project might be worth productionizing. These are a kind of skillset that researchers will need to develop in order to make research software systems go well.

These points concern how the research software system brings organizational work to bear on the problem of unplannability, but there is another sense in which the concept helps us understand the researchers’ relationship with unplannability. That is that it highlights the

importance of retrospection in the exploratory process. The process of running an open-ended test is a process of improvisation (Weick, 1995). It relies on the ability to assess the outcomes of past efforts rather than adhering to the projections of a protocol or plan. On the one hand this is a shift in perspective from the forward-looking predictive model of science, in which the movements that matter are the grand, prophetic, and falsifiable hypotheses or claims (Lakatos, 1989; Popper, 2014 [1962]). It is not my goal to examine that shift or its implications here except to point out that the pattern of tests and retrospection that I describe are not situations where clear, well-formed hypotheses can be mobilized ahead of time. I would hold with Fleck (1979 [1935]) that the experiments where the outcomes can be defined as a simple “yes” or “no” are exceptions in a much longer trajectories of exploratory testing and poorly formed hypotheses. In looking only at clearly-defined hypothesis driven tests, then, we miss out on a great deal of scientific sensemaking, and that is part of what I have tried to retrieve here. In this sense part of my argument is that scientists already have developed methods for working through situations of unplannability (indeed it is their bread and butter), although, as I argue in section 8.3 both software engineers and researchers have opportunities to learn new things about these methods as they might implemented in software development and use.

In a more modest way, the acknowledgement of retrospection can also be supported in practical ways. This could be done through the development of tools such as those that render code change histories in notebooks (Head et al., 2019; Kery and Myers, 2018). Thinking of scientific work in terms of specific ‘runs’ or workflows (Goble et al., 2010) could provide a context for associating specific versions of code with specific outputs, as well as the more general use of identifiers like Git hashes (Aklaghi et al., 2021; see also Stodden, 2020), which nail down the specific version of code that was used. An important point to be made with regards to this possibility is that the use case for one researcher or a small group of researchers hoping to make sense and keep track of past tests is quite different from the use case of demonstrating or

reproducing established findings for the purpose of review (Feinberg et al., 2020). This difference may play out in a number of ways, but one is in the level or immediacy with which retrospection may need to happen. As Kery et al. (2017) point out, the exploratory programming process needs an “informal” and short term kind of versioning that typical Git versioning does not afford particularly well. Their system works with branches and commits in the typical code versioning sense, but those concepts refer to snippets of code within a notebook rather than whole files or codebases.

While these kinds of tools may be quite helpful in supporting retrospection, it is also something that can be worked on through the design of certain kinds of meetings and spatial interactions. The data rampage, for instance, is a crucial space that the Radio Group uses for retrospection in particular ambiguous parts of the research process, and while it leverages the production of particular kinds of plots it relies just as much on their configuration in a room and the phases and rules of discourse the group uses in the process.

These observations on retrospection and exploratory process bring us back also to the issue of problem formation. My examination in Chapter 4 uses problem formation as an analytical route into an exploratory process. In other words, by looking at how problems come to take shape, rather than starting the examination when the problems are already well formed, we are able to pull out hugely important aspects of the Radio Group’s work, which are distinct from processes of software production and could be supported (or break down) in their own ways. This bolsters the value of problem formation as an analytical tool. While the benefits of a focus on problem formation are clear in the scientific context, I would argue that they would have value also in examining industrial contexts of software work and particularly contexts of data analysis. Indeed, this is a context in which the notion of “problem formulation” is re-emerging (Passi and Barocas, 2019; Passi and Sengers, 2019).

My approach also situates problem formation in terms of its material practice. Part of the benefit of the word “formation” in this usage is that it does not imply a verbal or grammatical mode. While the verbal statements of problems are hugely important, there are a variety of material objects and representations that people interact with in forming a problem, from the plots described here to design drawings (Ewenstein and Whyte, 2009) to prototypes (Vinck, 2011). Problem formation is certainly about changing people’s understandings of a situation, but they do so through interactions with the world around them, and “formation” is an apt word to capture this aspect of the process. Some aspects of these material interactions are captured in prior considerations of problem formation, such as in Fujimura’s (1987) reference to the outcomes of tests in a wet lab. However, closer attention to these kinds of material interactions, which are common in studies of computer supported cooperative work, could be a valuable way of deepening analyses of problem formation.

8.3 Sociotechnical change in software practice

It is well known that many fields are undergoing significant changes around software and software development work, but the exact nature of these changes is often discussed in broad terms or taken implicitly. In part this is because not much work has focused on the process of adoption or integration of software engineering practices at all. The focus has been instead on codifying best practices and on identifying barriers. This dissertation certainly does not put forward a new model of technology adoption or sociotechnical change writ large, but the case examined here presents an opportunity to do some specifying work about processes of change. The points that I will make generally fall under a notion of the integration and adaptation of software engineering practices rather than their adoption.

One primary consideration is that the kinds of changes that research groups or disciplines are undergoing is situated in a historical engagement with software and development work. Some

fields are taking up bespoke programming as novel activity altogether, others are shifting from old languages to new ones (Python or R in many cases), others are looking to adopt software engineering practices on top of programming practices that are long-established in their field (Sutherland et al., 2025), and yet others are undergoing broad professional changes as research software engineers become a part of the research landscape (Sims, 2022; Baxter et al., 2012; Berente et al., 2017). Most cases are likely a mix of these things. For the most part this study has focused on the adoption of software engineering practices into a community already largely familiar with programming. The adoption of software engineering practices is of course the focus of most of the recent discourse around scientific software, but these other considerations create differences between cases because research groups are starting in different places and pursuing different goals in changing their work around software.

This connects with the larger point that the adoption of software engineering practices can sometimes be taken as a generic process when in fact it will be highly tailored to the situation of a given research group or collaboration. Prior work has outlined a number of characteristics as common or essential in some way to the scientific context as it relates to software work. These include the idea that scientists are often not formally trained in programming or software development (Heaton and Carver, 2015), that the reward structures of science favor discovery over toolmaking (Du et al., 2021), and that they have a hard time establishing requirements and testing oracles (Kanewala and Bieman, 2014), among others. These are valuable guides to the cultures and landscapes of the sciences, but it is also important to consider the dimensions along which research groups and collaborations might vary. Sutherland et al. (2025) illustrate how two research groups take quite different approaches to integrate software engineering practices into their work, and these different approaches entail different organizational strategies and outcomes. Howison and Herbsleb (2011) have disaggregated the incentives that researchers have towards building software, differentiating between, for instance, researchers

writing software that incidental to their research work versus researchers in a software-focused subfield, who may receive some direct academic credit for their development work. They too highlight that cases may be hybrids of the different orientations they describe.

Enumerating all the different things that might shape a research group's engagement with software engineering practices is likely a sisyphean task, but there are some usual suspects. Funding is a factor, both for being able to support work focused on software production but also for the ability to hire research software engineers whose work focuses primarily on software development. This may emerge as a matter of the resourcing of one's institution, in that some institutions are more likely to be able to support dedicated research software engineering teams that they might be able to work with. Established practice around software is another factor. The Radio Group works in a field where programming, if not software engineering, has been an aspect of day-to-day research for a long time, such that the head of the group worked extensively with Fortran when he was a Ph.D. student. Closely connected with this is the legacy systems that the group might be working with. One of the Radio Group's primary analysis pipelines was written in IDL because that was the language that the younger members of the group were familiar with at the time that they started working on. IDL had been popular in physics for a certain period of time before it required licensing. As I described elsewhere, the presence of this legacy system shapes how they engage new tools (Sutherland et al., 2024). Moreover, it affects their ability to refactor or retroactively apply new practices. Although members of the group said that the software was too large and complicated to rework anyway, the absence of easy testing systems in IDL did not help.

The integration of software engineering practices into scientific work is likely to be selective rather than universal across all contexts in which researchers work with software. This is a direct assumption of the notion of the research software system, and it aligns with points that

have appeared in software engineering scholarship on scientific software. Judith Segal makes this point clearly in her examination of scientific software development:

> "What I am saying here is that there are at least some situations in which software engineers should not try to impose the full machinery of traditional software engineering on scientific software development. It is not the case that scientists developing their own software in the contexts discussed above are able coders but totally undisciplined, as was once said to me by a software engineer. They are just as disciplined as the context demands" (Segal, 2008b).

Segal leverages the notion of context here in the way that I develop regimes of a research software system. They define distinct spaces where different expectations and criteria for software work rule.

The integration of software engineering practices is also unlikely to be a bilateral process, where individual researchers encounter the existence of practice, as it is described in writing, and then take it up in their personal work. Some prior work has framed uptake instead as a matter of collective integration and adaptation of practices (Easterbrook and Johns, 2009), and these are the kind of dynamics I identify as well. A key point here is that there is a complex sociality to the sharing and integration of new practices. Actions like championing and demonstration involve not only the uptake of some practice but the performance of it in ways that are seen by others and responded to in an ongoing interactional negotiation of good, proper, or useful ways of working. Part of the significance of this is simply the interpersonal tensions that can arise around these interactions, and considering them explicitly in efforts to navigate change around software practice. It is also an important consideration in any effort to understand diffusion of technologies or techniques.

Another aspect of the sociality of adoption is that it should turn out attention to processes of boundary spanning. Many members of the CDA referenced Mila's presence at a data science institute as an early source for the kinds of software engineering practices that they took up.

This process of specific people bringing practices from one context and adapting them and demonstrating them in another context is an important place to look to understand how software engineering practices might actually get taken up and integrated successfully into scientific communities. Moreover, the demonstration and sharing of such practices is a potentially critical aspect of the work of research software engineers.

Another consideration is that the uptake of software engineering practices can have constitutive change to scientific work itself. This may seem obvious, but the lion's share of the literature on scientific software has been focused on how to promote or facilitate the uptake of software engineering practices, rather than on what kinds of changes these practices might have on scientific work once adopted. There are usually understandings embedded in these discussions that adopting engineering practices with *expedite* science: it will be roughly the same kind of science but there will be faster and more correct results (Wilson et al., 2006). There are, alternatively, propositions that science will be radically transformed by increases or innovations in software and data technologies (Djorgovsky, 2005; Brescia, 2017). Studies of escience and cyberinfrastructure have looked at the shaping of science by tools such as bespoke software (Paine and Lee, 2015) and databases (Hine, 2006). This of course draws on broader and longer-running conceptions of science as practice (Pickering, 2010; Knorr, 1979).

If we take the idea that software constitutes the working objects of science, as I have argued here, we must lean into this latter view of the entanglement of 'science' with software practice and look at the ways that changes in software development practice change 'the science' in one way or another. This is a huge topic that could run a broad range of understandings in what constitutes 'the science.' One route into this consideration is the one that I have taken here, which has to do with the workability (Spencer, 2015) of software as a tool for epistemic work. In this understanding software has materiality in both use and development (and patterns of iterative use and development), which is captured in terms like viscosity (Green and Petre,

1996), brittleness (Spencer, 2015), kleenex code and so on. This materiality affects the way that novel phenomena can be rendered and apprehended (perhaps detected) by the researcher but also the interactivity that the researcher is able to establish with these phenomena. As I discussed in Chapter 4 this includes the rate at which they are able to develop new probes and see new outcomes. Examining knowledge turns (Goble et al., 2013), for instance, would be one route into this understanding of 'the science.' The recognition of software as partially constitutive of scientific practice should open the door to many more kinds of considerations of how science changes with the change in software practice, from citation networks to professional identity. These possibilities will be discussed further in the next section.

There is one last point worth making about the efforts that members of the Radio Group made in changing their practices around scientific software, which is that it takes a great deal of reorganizing to accomplish changes in the materiality of a few large codebases. As MacKenzie (2006) argues, code can certainly be wildly mutable, but in particular understandings of the materiality of changing it and extending it over time it can be almost immovable, requiring huge efforts and significant changes in the way work is done in order to accomplish a sense of agility or flexibility. This is perhaps a familiar problem in industrial contexts, but it does problematize the idea of minimal changes in practice as the route forward for change in scientific software work. Best practices are certainly useful entry points and may be enough for some groups, but there is a larger problematic of reorganizing around software that some collaborations will need to engage. Longer running and more laborious relationships with legacy software are another sphere of concerns research communities need to navigate (Cohn, 2016).

8.4 Reapproaching scientific software

Scientific software is a topic that should interest both those interested in processes of software engineering as well as those interested in processes and material practices of the sciences.

Part of my goal has been to make the category of scientific software visible to both of these groups in new ways. In this section I will discuss the category of scientific software and some connections to these two broad disciplinary interests. One goal of this is to describe the extent and potential usefulness of the sensitizing concepts described here.

The first boundary that should be addressed is the one between scientific activity and non-scientific activity. This study has focused on exploratory work in the sciences, and the sciences are one place where the implications of my findings might be most useful, but it is relevant to a much broader set of activities that involve a large degree of exploration using software tools. There are many of these kinds of activities in the world. Kery and Myers (2017) have discussed a number of these, including the creation of art through programming, learning programming itself, and more generally in software design and development. Anyone who has programmed has likely performed small open-ended tests aimed at learning how something works rather than at building enduring software. In this sense the project of science is not a special ontological category where special kinds of sensemaking and development activities exist. The work of the Radio Group likely diverges from many software production contexts in the degree and duration of exploratory work that they engage in, or in the amount of time that they must spend working with incomplete epistemic objects[7]. Moreover, particular situations in the sciences may be unique in the sense of everything being uniquely situated in its historical context, but they do not constitute fundamentally discrete categories of action.

It is important to note, however, that the sensitizing concepts that I have developed here point our attention specifically to situations where there is a great degree of exploratory work. The notion of the research software system points in particular to the bounding and articulation of

[7] There have been many efforts to demarcate a special logic or process for science of course, but there are also many designations of science as being different by degree in one respect or another. Polanyi (1945) does this with the degree of “autonomy” of the profession of science and Nersessian (1988) does something similar with scientific reasoning.

different regimes of practice. This might be a useful guide in situations where exploratory, open-ended testing work is significant enough that it is worth designating as a distinct regime of practice. For instance, Middleton et al.'s (2020) study of sports analytics presents some similarities in that analysts need the flexibility in changing research questions and exploring the significance of different analyses. Communities such as these may need to maintain space for a great deal of exploratory work, and consequently need to manage the articulation of production and exploration processes. Where such an articulation has not happened, the notion of the research software system might point one's attention to how it could be done. Passi and Sengers (2020) have described the need for greater attention to problem formulation in data science contexts, and consideration of the research software system in those contexts might point one to building out dedicated contexts explicitly oriented towards figuring out what it is that the system should be doing, as well as how that work connects with the work of production and implementation.

This relation to software production can be seen by comparison with Agile methodology. Easterbrook and Johns (2009) have outlined both similarities and differences between the way scientists develop software and Agile (citing once again the absence of oracles in verification processes). I also see similarities between Agile and the process of exploratory programming that I develop here, particularly in its amenability to changing requirements and its focus on rapid implementation and 'seeing what happens.' One point of difference is that Agile, as a whole, is oriented towards production. A tenet of the methodology is that the primary measure of progress is working software (Beck et al., 2001). The process of exploration I have outlined here is focused primarily on the development of new capacities, which involves software, but is not defined solely by working code. It also encompasses new understandings of interactional possibilities. It also does not immediately demand robust and reliable, well-tested code.

This could set up a distinction in which software is used on the one hand to learn new things and on the other hand to build products that 'do' things in the world. This might put activities like science and learning to program on one side and industrial software development of the other. This might be a useful distinction in some cases, but it does not hold up well in my model of the research software system. Firstly, as I argued in Chapter 4, running even the most robust software in new contexts has the potential to generate new problems. For this reason, any kind of design and development work involves learning new things about the interactional possibilities of the product under development. While I am primarily talking about the capacities of instruments in this case as well as the development of new understandings of scientific objects such as the 21cm signal, any effort to build a software system might involve learning new things about the people who use it, its objectives, and the possibilities of its intervention in the world. Agile is designed precisely with this in mind. It assumes that the developer does not already know everything about what the system should do at the outset of development.

There is also, however, a very important sense in which learning new things relies on some concern or attention to the production of one's tools and materials. My understanding of the process by which researchers figure out new things is one embedded in their concrete interactions with a situation, and in particular with systems of instrumentation. In other words, in learning about a phenomenon they are learning about their own potentials for interaction with that thing, how they can produce it, interact with it, use it for other purposes, and how it responds to different kinds of probes. The researcher learns about a new phenomenon as an interlocutor in a "conversation with the materials," so to speak (Schön, 1983). Greater attention to the production of software tools in this sense refines the researcher's ability to pursue that conversation, to define better tests and to understand their outcomes. This is another point which is embedded in the notion of the research software system, and it is what undergirds the

argument above about the re-organization of development work contributing to the ability of researchers to accomplish flexibility in technoscientific change.

Both of these points help break down a harsh distinction between contexts of industrial software production and scientific research. The distinction that remains is that researchers are likely to spend a great deal of time with highly incomplete and uncertain epistemic objects, and that their activities emphasize learning new things about the interactional possibilities with regards to these uncertain entities.

This discussion concerns the issue of whether science is really a special kind of work with software, but we can also reapproach software as a topic relevant to social studies of science. There have of course been a great many investigations in science studies which take as their object things that are software (Lynch and Edgarton, 1987). The issue here really has more to do with what software development or software engineering specifically might mean to social studies of science, rather than just the artifact of software itself. Specifically, the thing that matters in the discourse that has recently arisen around scientific software is a kind of end-user development situation, in which writing code and reusing it is a primary way that a researcher engages their objects of study.

There are some ways in which this situation can become more visible in accounts of scientific work and culture. For instance, Baneke (2023) places the contemporary ‘big science’ mode of astronomy at the historical conjunction of the instrument cultures of optical astronomy, space science, and radio astronomy. A historical account such as this rightfully comes to focus around the large hardware components that shaped these particular traditions. More and more, however, our accounts of instrument cultures will need to account for software tools as points of confluence of communities of researchers as well as potent agents in shaping the kind of science that gets done. Such analysis can certainly look at readymade applications that come to

be used in scientific contexts, such as Excel (Vertesi, 2020), but there are already long traditions that have formed around tools designed by and for researchers in particular communities and fields. These software components are not only common resources but also in many cases common investments and matters of collective design.[8] This kind of project has already begun around data and data formats (Scroggins and Boscoe, 2020). For instance, Strasser's (2019) examination of the shifting moral economy of data creation and access in biology takes data as a matter of common investment as well as being at the center of changing traditions of experimental work.

Software is also becoming extremely significant in scientific work as a matter of profession. The notion of research software systems is not designed to address this issue, but some of my findings do raise some questions about it. For instance, my observations capture programming as a constitutive skillset for doing work in Radio Group and the larger collaborations it engaged in. In exploratory work this was primarily a matter of being extremely capable with data manipulation and mathematical tools and perhaps most importantly being proficient in plotting packages in Python. However, amongst research scientists in software production contexts there was substantial interest in new tools for code formatting, unit testing, or package distribution (tools for distributing code as a self-contained package to users). These tools were more relevant to the kind of software engineering that is usually considered as ancillary to scientific concerns. At least in this case, the people who were most savvy in finding and adapting these kinds of techniques were research scientists, in part perhaps because more senior members tried to protect Ph.D. students from having to do a great deal of "IT work" (Sutherland et al., 2024).

[8] In the context of radio astronomy the AIPS software (Wells, 1985) is one example of a long-running and commonly used tool. Such tools interact with the communities that develop around hardware instrumentation, as, for instance, AIPS was developed around the Very Large Array (VLA) but with a generality in design that was intended for a larger radio astronomy community (Greisen, 2003).

These observations represent certain shifts in the content of the material practice of science. Galison (1997) points out how physicists working with bubble chambers and cloud chambers at different points in time would need to develop understandings of the essential materials of their work: the plastic lexane and its qualities, film and photography, or how to harden electronics against radiation (pg. 8). In a similar way members of the Radio Group increasingly find the need to understand unit tests and packaging systems, the use of deprecation and the benefits and dangers of computational notebooks. Part of the significance of my argument about the use of production code in exploration is that these assessments play into researchers' assessments of the outcomes of tests. It is part of what makes up their professional vision (Goodwin, 1997) in the estimation and explanation of particular features on a plot, but also in subtle assessments of the quality of a software tool as a reliable instrument.

Another issue raised by recent movements towards research software engineering (RSE) is the role of the RSE in the research lab or university. The work of technicians has a number of associations in prior literature, such as being a point of communication between a scientists and the empirical content of an instrument (Barley and Bechky, 1994): they do the reading and assessment of whether a tool is working properly or discarding outcomes as anomalous. Technicians are also often invisible subordinates in the research lab (Shapin, 1989), as the association goes. In my case the work of software engineering was taken on entirely by researchers themselves, but as described the typical researcher at the research scientists level became partially embedded in the world of software engineering. The professional identities in that space remain under construction (Berente et al., 2017). One member of the CDA, who was a frequent champion of software engineering practices, said that they were familiar with research software engineering as a movement but could not associate with it well because that movement had chosen the word "engineer" as the descriptor, which they did not align with. It also seems unlikely that software engineers entering the sciences from software industries will

take a role typically associated with the technician. The way these identities develop will, however, shape relations central to knowledge production.

8.5 Conclusion and limitations

This study has developed the notion of research software systems both as a way of intervening in a long-running discourse around research software, and as a way of inspecting software as a tool for exploratory research work. The concept has promise as a way of understanding all kinds of exploratory work with software, and in considering the many different ways that software might be used as a research tool. There are, however, a couple of blindspots to the concept as it is developed here, which would make excellent points of departure for future work.

The first of these is that while I develop a notion of heterogeneity in the concept of a *system*, this case does not more deeply explore the extent of heterogeneity of software as it could. There is also, for instance, the notion of scientific software ecosystems (Howison et al., 2015), which presents its own consideration of heterogeneity. These two concepts are doing something slightly different in that the notion of research software systems is preoccupied with the epistemic role of a larger software system from the perspective of a specific research group, whereas the ecosystems notion is focused on the broader landscape of producers, providers, stewards, and funders. Nevertheless, it would be productive to pursue what possible new questions open up when juxtaposing these concerns.

Another area opened but not engaged deeply by this study is the notion of instruments as ensembles. The notion of a research software system forces the idea that the ‘instrument’ used in a given test is in fact a vast network of associated actants, which hold together or do not hold together. I also point to the notion of modularity as one place where the struggle with keeping this vast array in order plays out. However a much more developed theorization of this dynamic is needed. While the notion of instrument points our attention to certain epistemic aspects of

tools, the notion of instrument itself tends to move towards breaking down into concepts more like machineries (Monteiro, 2022) or infrastructures (e.g. Edwards, 2013). Of particular interest to the notion of research software systems is where the involvement or investment of the researcher leaves off in this vast infrastructure. In some ways the researcher cannot ignore any aspect of that vast system, but in others there are practical dropoff points where the software used falls out of the scientific endeavor and into the concerns of industries or open source communities. My conception also, for reasons of scope, also leaves off at the edge of the software system. McCray (2014) points out how already in the 1970s the hardware instruments of astronomy were becoming hybrids of hardware, software, and databases. It would be valuable to approach the tools of science as hybrids and not strictly as software components.

This introduces another important limitation of the concept of the research software system. The research software system forefronts software as an object of concern and as an object of work and negotiation. In doing so it parallels the concerns of the Radio Group, the CDA, and much broader discourses in the sciences (e.g. Joppa et al., 2013). A risk that comes with this focus is the naturalization of “software” as an innate category of the world, which entails its own kinds of activities and processes. Such a view would suggest that between the 1970s and the present historical moment the sciences have transitioned from one fundamental category of activity to a software activity, and software now defines what science is and how it proceeds. Something like this animates discussions of data-oriented revolutions in the sciences, where the use of large datasets is posited as a “fourth paradigm” of science (Hey, Tansley, and Tolle, 2009), progressively succeeding prior modes of scientific inquiry.

The problem with this is that software, and a few specific ways of working around it, come to *stand in* for a diversity of complex hybridities that actually constitute the sciences. Taking scientific work as a research software system in a narrow sense could flatten the activities of science to simply being software processes, and reduce even those processes to a few

prescribed by industrial contexts of software production. This could further designate authority and decision making in the sciences to those who have expertise in particular ways of working around one object of science, its software. In other words, the notion of the research software system puts software in focus, where it has often gone unobserved, but it runs the risk of reducing scientific activity if that focus is taken as an immovable characteristic of the world rather than an aspect of my methodological approach.

There are a couple remedies for this problem. The first is to take the concept as a sensitizing concept, as I present it. The research software system would then not define science holistically in terms of software-related activities or categories, but rather provide a lens for inspecting particular kinds of material practices that are increasingly essential to scientific work in a variety of fields. One would not expect to find the same system in different cases. There are other ways of using these findings that would turn the research software system into a more nimble conceptual tool. For example, one thing that characterizes the concept as I have developed it here is as a re-situation of production processes in a larger heterogeneous arrangement, which also considers exploration. This is an approach that could be taken in other contexts where a production process is the implicit assumption, regardless of whether they revolve around software. In other words, the research software system might point one towards software, but it also might point one towards exploration or problem formation, the interaction of these with production work, or towards the articulation of these different processes.

This latter approach can be pursued by taking the findings here into cases beyond software. What can be learned, for instance, by looking at continuities and divergences between research software systems and production and exploration with other kinds of objects, such as telescopes, centrifuges, and thermometers? Sepkoski (2017) takes an approach along these lines in addressing the database in paleontological work, describing practices for compiling and organizing data prior to the general uptake of computers in paleontology. The goal of such an

approach is not anachronism, applying concepts developed for software and data systems to practices employed prior to the ‘computer era.’ Rather the goals are a) taking software and other objects under a larger category of interest such as modes of collection and mobilization of evidence (in Sepkoski’s case), and b) accounting for how communities of people integrate and adapt new technologies into existing practices and understandings (which often change in the process), rather than taking on ways of working that are somehow innate to the materiality of an object. Such an approach would present the research software system as entangled with other aspects of scientific work and other scientific tools, and it would also recognize variety in how research software systems might be enacted across different fields and disciplines with different histories of material practice.

A last point is that this case presents perhaps an exemplary case of an end-user development scenario, but this may not be a good representation of the wider variety of research software systems. In the Radio Group and their larger collaborations software work was taken on by researchers who were still for the most part focused on scientific outcomes. However, many other fields are finding preferred arrangements through the delegation of work amongst research-focused work and the more software-focused work of research software engineering (e.g. Sutherland et al., 2025). Studies of the dynamics of coordination and professionalization are certainly important routes forward on that topic, but it would also be worth examining the different configurations a research software system takes in the various delegations of work that might play out with the growth of research software engineering. This is a way of keeping our eye on the relationship between the organization of the lab and collaboration and the dynamics of epistemic work.

References

Ahalt, S., Band, L., Christopherson, L., Idaszak, R., Lenhardt, C., Minsker, B., ... & Zimmerman, A. (2014). Water Science Software Institute: Agile and open source scientific software development. *Computing in Science & Engineering*, *16*(3), 18-26.

Akhlaghi, M., Infante-Sainz, R., Roukema, B. F., Khellat, M., Valls-Gabaud, D., & Baena-Galle, R. (2021). Toward long-term and archivable reproducibility. *Computing in Science & Engineering*, *23*(3), 82-91.

Ankeny, R. A., & Leonelli, S. (2016). Repertoires: A post-Kuhnian perspective on scientific change and collaborative research. *Studies in History and Philosophy of Science Part A*, *60*, 18-28.

Ashforth, B. E., & Fried, Y. (1988). The mindlessness of organizational behaviors. *Human Relations*, *41*(4), 305-329.

Baker, K. S., & Bowker, G. C. (2007). Information ecology: open system environment for data, memories, and knowing. *Journal of Intelligent Information Systems*, *29*, 127-144.

Baldwin, C. Y., & Clark, K. B. (2000). *Design rules, Volume 1: The power of modularity*. MIT press.

Baneke, D. (2023). Big Astronomy: large telescopes and the dual narrative of impact. In *Big Science in the 21st Century: Economic and societal impacts* (pp. 28-1). Bristol, UK: IOP Publishing.

Barley, S. R., & Bechky, B. A. (1994). In the backrooms of science: The work of technicians in science labs. *Work and occupations*, *21*(1), 85-126.

Basili, V. R., Carver, J. C., Cruzes, D., Hochstein, L. M., Hollingsworth, J. K., Shull, F., & Zelkowitz, M. V. (2008). Understanding the high-performance-computing community: A software engineer's perspective. *IEEE software*, *25*(4), 29.

Baxter, R., Hong, N. C., Gorissen, D., Hetherington, J., & Todorov, I. (2012, September). The research software engineer. In Digital Research 2012.

Beath, C. M. (1991). Supporting the information technology champion. *MIS quarterly*, 355-372.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S. J., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for Agile Software Development. Agile Alliance.

Berente, N., Howison, J., Cutcher-Gershenfeld, J., King, J. L., Barley, S. R., & Towns, J. (2017). Professionalization in cyberinfrastructure. Available at SSRN 3138592.

Berliner, P. F. (1994). *Thinking in jazz: The infinite art of improvisation*. University of Chicago Press.

Bernal, J.D. (1939) The Social Function of Science. George Routledge and Sons, Ltd.: London.

Bietz, M. J., & Lee, C. P. (2009). Collaboration in metagenomics: Sequence databases and the organization of scientific work. In *ECSCW 2009* (pp. 243-262). London: Springer London.

Bietz, M. J., & Lee, C. P. (2009). Designing Cyberinfrastructure for Future Users. *International Reports on Socio-Informatics (IRSI)*, 6(4), 16-23.

Bietz, M. J., Baumer, E. P., & Lee, C. P. (2010). Synergizing in cyberinfrastructure development. *Computer Supported Cooperative Work (CSCW)*, *19*, 245-281.

Bietz, M. J., Ferro, T., & Lee, C. P. (2012, February). Sustaining the development of cyberinfrastructure: an organization adapting to change. In *Proceedings of the ACM 2012 conference on computer supported cooperative work* (pp. 901-910).

Bloor, D. (1991). *Knowledge and social imagery*. University of Chicago Press.

Blumer, H. (1986). *Symbolic interactionism: Perspective and method*. Univ of California Press.

Boujut, J. F., & Blanco, E. (2003). Intermediary objects as a means to foster co-operation in engineering design. Computer Supported Cooperative Work (CSCW), 12(2), 205-219.

Bowker, G. C. (1994). *Science on the run: Information management and industrial geophysics at Schlumberger, 1920-1940*. MIT press.

Brand, S. (1995). *How buildings learn: What happens after they're built*. Penguin.

Brescia, M., Cavuoti, S., Amaro, V., Riccio, G., Angora, G., Vellucci, C., & Longo, G. (2017, October). Data Deluge in Astrophysics: Photometric Redshifts as a Template Use Case. In International Conference on Data Analytics and Management in Data Intensive Domains (pp. 61-72). Cham: Springer International Publishing.

Bridgman, P. W. (1947). Science and freedom reflections of a Physicist. *Isis*, *37*(3/4), 128-131.

Bryant, A., & Charmaz, K. (2007). Grounded theory in historical perspective: An epistemological account. *The SAGE handbook of grounded theory*, 31-57.

Burrell J (2016) How the machine 'thinks': Understanding opacity in machine learning algorithms. Big Data and Society 3(1): 1–12.

Callon, M. (1998). An essay on framing and overflowing: economic externalities revisited by sociology. *The sociological review*, *46*(1), 244-269.

Cambrosio, A., & Keating, P. (1988). "Going monoclonal": Art, science, and magic in the day-to-day use of hybridoma technology. *Social Problems*, *35*(3), 244-260.

Camus, A., & Vinck, D. (2019). Unfolding digital materiality. How engineers struggle to shape tangible and fluid objects. *DigitalSTS: A Fieldguide for Science & Technology Studies*, 17-41.

The Carpentries (2021) The Carpentries. https://carpentries.org/

Charmaz, K. (2014). Constructing grounded theory (introducing qualitative methods series). *Constr. grounded theory*. Sage.

Clarke, A. E., & Star, S. L. (2008). The social worlds framework: A theory/methods package. *The handbook of science and technology studies*, *3*(0), 113-137.

Cohn, M. L., Sim, S. E., & Lee, C. P. (2009). What counts as software process? Negotiating the boundary of software work through artifacts and conversation. *Computer Supported Cooperative Work (CSCW)*, *18*, 401-443.

Cohn, M. L. (2016, February). Convivial decay: Entangled lifetimes in a geriatric infrastructure. In *Proceedings of the 19th ACM conference on computer-supported cooperative work & social computing* (pp. 1511-1523).

Cohn, M. L. (2019). Keeping software present: Software as a timely object for STS studies of the digital. *DigitalSTS: A field guide for science & technology studies*, 423-446.

Cohen, M. D. (2007). Reading Dewey: Reflections on the study of routine. Organization Studies, 28, 773–786.

Colfer, L. J., & Baldwin, C. Y. (2016). The mirroring hypothesis: theory, evidence, and exceptions. *Industrial and Corporate Change*, *25*(5), 709-738.

Collins, H. (1992). *Changing order: Replication and induction in scientific practice*. University of Chicago Press.

Collins, H. (2010). Gravity's shadow: The search for gravitational waves. In *Gravity's Shadow*. University of Chicago Press.

Conway, M. (1968) How do committees invent? Datamation. 14(4) 28-31.

Carver, J. C., Weber, N., Ram, K., Gesing, S., & Katz, D. S. (2022). A survey of the state of the practice for research software in the United States. *Peerj computer science*, *8*, e963.

Cataldo, M., Bass, M., Herbsleb, J. D., & Bass, L. (2006). Managing complexity in collaborative software development: on the limits of modularity. *Supporting the Social Side of Large Scale Software Development*, 15.

Cataldo, M., & Herbsleb, J. D. (2008, November). Communication networks in geographically distributed software development. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work* (pp. 579-588).

Cleland, C. E. (2001). Historical science, experimental science, and the scientific method. *Geology*, *29*(11), 987-990.

Cleland, C. E. (2002). Methodological and epistemic differences between historical science and experimental science. *Philosophy of science*, *69*(3), 474-496.

Cohen, Jeremy, Daniel S Katz, Michelle Barker, Neil Chue Hong, Robert Haines, and Caroline Jay. 2020. The four pillars of research software engineering. IEEE Software 38, 1 (2020), 97–105.

Cohn, M. L. (2016, February). Convivial decay: Entangled lifetimes in a geriatric infrastructure. In *Proceedings of the 19th ACM conference on computer-supported cooperative work & social computing* (pp. 1511-1523).

Cohn, M. L. (2019). Keeping Software Present: Software as a Timely Object for STS Studies of the Digital. *DigitalSTS: A field guide for science & technology studies*, 423-446.

Cohoon, J., & Howison, J. (2021). Norms and open systems in open science. *Information & Culture*, *56*(2), 115-137.

D’Adderio, L. (2003), ‘Configuring Software, Reconfiguring Memories: The Influence of Integrated Systems on the Reproduction of Knowledge and Routines’, Industrial and Corporate Change, 12(2): 321–350.

D’adderio, L. (2008a). The performativity of routines: Theorising the influence of artefacts and distributed agencies on routines dynamics. *Research policy*, *37*(5), 769-789.

D’Adderio, L. (2008b), ‘The Dependable Transfer of Routines and Capabilities: A Performative View’, Academy of Management (AOM) Meeting, Paper No. 15238, Anaheim, CA.

D’adderio, L. (2011). Artifacts at the centre of routines: Performing the material turn in routines theory. *Journal of institutional economics*, *7*(2), 197-230.

Darch, P., Carusi, A., & Jirotka, M. (2009, December). Shared understanding of end-users' requirements in e-Science projects. In 2009 5th IEEE international conference on e-science workshops (pp. 125-128). IEEE.

Daston, L., & Galison, P. (2007). Objectivity.

Daston, L. (2014). 16 Beyond Representation. *Representation in scientific practice revisited*, 319.

De Boer, B. (2021). *How scientific instruments speak: Postphenomenology and technological mediations in neuroscientific practice*. Rowman & Littlefield.

DeBoer, D. R., Parsons, A. R., Aguirre, J. E., Alexander, P., Ali, Z. S., Beardsley, A. P., ... & Cheng, C. (2017). Hydrogen epoch of reionization array (HERA). Publications of the Astronomical Society of the Pacific, 129(974), 045001.

Dew, K. N., Landwehr-Sydow, S., Rosner, D. K., Thayer, A., & Jonsson, M. (2019). Producing printability: Articulation work and alignment in 3d printing. Human–Computer Interaction, 34(5-6), 433-469.

Dewey, J. (2023 [1938]) Logic: The Theory of Inquiry. Affordable Classics.

Djorgovski, S. G. (2005). Virtual astronomy, information technology, and the new scientific methodology. In Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05) (pp. 125-132). IEEE.

Dourish, P. (2022). *The stuff of bits: An essay on the materialities of information*. MIT Press.

Du, C., Cohoon, J., Priem, J., Piwowar, H., Meyer, C., & Howison, J. (2021, October). CiteAs: better software through sociotechnical change for better software citation. In *Companion Publication of the 2021 Conference on Computer Supported Cooperative Work and Social Computing* (pp. 218-221).

Easterbrook, S. M., & Johns, T. C. (2009). Engineering the software for understanding climate change. *Computing in science & engineering*, *11*(6), 65-74.

Edwards, P. N. (2013). *A vast machine: Computer models, climate data, and the politics of global warming*. Mit press.

Ewenstein, B., & Whyte, J. (2009). Knowledge practices in design: the role of visual representations as epistemic objects'. *Organization studies*, *30*(1), 07-30.

Faulkner, R. R., & Becker, H. S. (2019). *" Do You Know...?" The Jazz Repertoire in Action*. University of Chicago Press.

Fedorova, M., Mazmanian, M., & Dourish, P. (2025). Coding Beauty and Decoding Ugliness: The Role of Aesthetic Concerns in Programming Practices. *Science, Technology, & Human Values*, *50*(1), 69-93.

Feinberg, M., Sutherland, W., Nelson, S. B., Jarrahi, M. H., & Rajasekar, A. (2020). The new reality of reproducibility: The role of data work in scientific research. *Proceedings of the ACM on Human-Computer Interaction*, *4*(CSCW1), 1-22.

Feldman, M. S., & Pentland, B. T. (2003). Reconceptualizing organizational routines as a source of flexibility and change. Administrative science quarterly, 48(1), 94-118.

Fleck, L. (1979). Genesis and Development of a Scientific Fact. Translated by F. Bradley & T.J. Trenn. Edited by T.J. Trenn & R.K. Merton. University of Chicago Press

Fujimura, J. H. (1987). Constructing ‘do-able' problems in Cancer research: Articulating alignment. Social studies of science, 17(2), 257-293.

Fujimura, J. H. (1992). Crafting science: Standardized packages, boundary objects, and “translation.”. Science as practice and culture, 168(1992), 168-69.

Galison, P. (1992) The many faces of big science. In *Big science: the growth of large-scale research.* Stanford University Press: Stanford, California.

Geiger, R. S., & Ribes, D. (2011, January). Trace ethnography: Following coordination through documentary practices. In 2011 44th Hawaii international conference on system sciences (pp. 1-10). IEEE.

Gersick, C. J. (1994). Pacing strategic change: The case of a new venture. *Academy of management journal*, *37*(1), 9-45.

Gerson, E. M. (2008). Reach, bracket, and the limits of rationalized coordination: Some challenges for CSCW. In M. S. Ackerman, C. A. Halverson, T. Erickson, & W. A. Kellogg (Eds.), Resources, Co-Evolution and Artifacts: Theory in CSCW (pp. 193–220). London: Springer.

Gibney, E. (2018) Astronomers detect light from the Universe's first stars. Nature news. www.nature.com/articles/d41586-018-02616-8

Gibson, J. J. (1979). The ecological approach to visual perception. Boston, MA: Houghton Mifflin.

Gieryn, T. F. (1983). Boundary-work and the demarcation of science from non-science: Strains and interests in professional ideologies of scientists. *American sociological review*, 781-795.

Gioia, T. 1988. The Imperfect Art. Oxford, New York.

Glaser V (2014) Enchanted algorithms: How organizations use algorithms to automate decision-making routines. In: Proceedings of the Annual Meeting of the Academy of Management, Philadelphia, PA.

Goble, C. A., Bhagat, J., Aleksejevs, S., Cruickshank, D., Michaelides, D., Newman, D., ... & De Roure, D. (2010). myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic acids research*, *38*(2), W677-W682.

Goble, C., De Roure, D., & Bechhofer, S. (2013). Accelerating scientists' knowledge turns. In Knowledge Discovery, Knowledge Engineering and Knowledge Management: Third International Joint Conference, IC3K 2011, Paris, France, October 26-29, 2011. Revised Selected Papers 3 (pp. 3-25). Springer Berlin Heidelberg.

Goguen, J. A. (1993, January). Social issues in requirements engineering. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering* (pp. 194-195). IEEE.

Goodwin, C. (1994). Professional Vision. *American Anthropologist*, *96*(3), 606-633.

Goody, J. (1987). *The Interface between the Written and the Oral*. Cambridge University Press.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, *7*(2), 131-174.

Greisen, E. W. (2003). AIPS, the VLA, and the VLBA. In *Information Handling in Astronomy-Historical Vistas* (pp. 109-125). Dordrecht: Springer Netherlands.

Grüning, B., Chilton, J., Köster, J., Dale, R., Soranzo, N., Van Den Beek, M., ... & Taylor, J. (2018). Practical computational reproducibility in the life sciences. *Cell systems*, *6*(6), 631-635.

Hayashi, S., Saeki, M., & Kurihara, M. (2006). Supporting refactoring activities using histories of program modification. *IEICE transactions on information and systems*, *89*(4), 1403-1412.

Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019, May). Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1-12).

Hey, T., Tansley, S., & Tolle, K. M. (2009). *The fourth paradigm: data-intensive scientific discovery* (Vol. 1). Redmond, WA: Microsoft research.

Hesse, M. (1980). Revolutions and reconstructions in the philosophy of science. Harvester Press.

Heaton, D., & Carver, J. C. (2015). Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, *67*, 207-219.

Hettrick, S., Antonioletti, M., Carr, L., Chue Hong, N., Crouch, S., De Roure, D. C., ... & Sufi, S. (2014). UK research software survey 2014.

Hirsch, S. L., Ribes, D., & Inman, S. (2022). Sedimentary legacy and the disturbing recurrence of the human in long-term ecological research. *Social Studies of Science*, 52(4), 561-580.

Hong, NC (2014) Minimal information for reusable scientific software. In Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2. 1).

Howison, J., & Bullard, J. (2016). Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature. Journal of the Association for Information Science and Technology, 67(9), 2137-2155.

Howison, J., & Crowston, K. (2014). Collaboration through open superposition: A theory of the open source way. *Mis Quarterly*, *38*(1), 29-50.

Howison, J., & Herbsleb, J. (2010). Socio-technical logics of correctness in the scientific software development ecosystem.

Howison, J., & Herbsleb, J. D. (2011, March). Scientific software production: incentives and collaboration. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work* (pp. 513-522).

Howison, J. and Herbsleb, J. D. (2013) Incentives and integration in scientific software production. In Proceedings of the 2013 conference on Computer supported cooperative work. 459–470.

Ihde, Don (1979) Technics and Praxis. Dordrecht: D. Reidel Publishing Company.

Jewett, T., & Kling, R. (1991). The dynamics of computerization in a social science research team: A case study of infrastructure, strategies, and skills. *Social Science Computer Review*, *9*(2), 246-275.

Jiménez, R. C., Kuzak, M., Alhamdoosh, M., Barker, M., Batut, B., Borg, M., ... & Crouch, S. (2017). Four simple recommendations to encourage best practices in research software. *F1000Research*, *6*, ELIXIR-876.

Jirotka, M., & Goguen, J. A. (Eds.). (1994). Requirements engineering: social and technical issues. Academic Press Professional, Inc..

Jirotka, M., & Luff, P. (2006). Supporting requirements with video-based analysis. IEEE software, 23(3), 42-44.

Joppa, L. N., McInerny, G., Harper, R., Salido, L., Takeda, K., O'Hara, K., ... & Emmott, S. (2013). Troubling trends in scientific software use. *Science*, *340*(6134), 814-815.

Karasti, H., & Blomberg, J. (2018). Studying infrastructuring ethnographically. *Computer Supported Cooperative Work (CSCW)*, *27*, 233-265.

Katz, D. S., & Chue Hong, N. P. (2018). Software citation in theory and practice. In Mathematical Software–ICMS 2018: 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings 6 (pp. 289-296). Springer International Publishing.

Katz, D. S., Hong, N. P. C., Clark, T., Muench, A., Stall, S., Bouquin, D., ... & Yeston, J. (2020). Recognizing the value of software: a software citation guide. F1000Research, 9.

Kanewala, U., & Bieman, J. M. (2014). Testing scientific software: A systematic literature review. *Information and software technology*, *56*(10), 1219-1232.

Kelly, D. (2015). Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. Journal of Systems and Software, 109, 50-61.

Kelly, D., D. Hook, R. Sanders (2009) Five recommended practices for computational scientists who write software, Computing in Science and Engineering 11, 48–53.

Kelly, D., & Sanders, R. (2008, July). The challenge of testing scientific software. In *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)* (pp. 30-36). Citeseer.

Kelly, D., Smith, S., & Meng, N. (2011). Software engineering for scientists. *Computing in Science & Engineering*, *13*(05), 7-11.

Kery, M. B., & Myers, B. A. (2017, October). Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 25-29). IEEE.

Kery, M. B., Horvath, A., & Myers, B. A. (2017, May). Variolite: Supporting Exploratory Programming by Data Scientists. In *CHI* (Vol. 10, pp. 3025453-3025626).

Kery, M. B., & Myers, B. A. (2018, October). Interactions for untangling messy history in a computational notebook. In *2018 IEEE symposium on visual languages and human-centric computing (VL/HCC)* (pp. 147-155). IEEE.

Kirschenbaum, Matthew G. *Mechanisms: New media and the forensic imagination*. MIT Press, 2008.

Klein, U. (2001). Paper tools in experimental cultures. *Studies in History and Philosophy of Science Part A*, *32*(2), 265-302.

Knorr, K. D. (1979). Tinkering toward success: Prelude to a theory of scientific practice. *Theory and Society*, *8*(3), 347-376.

Knorr-Cetina, K. D. (1981). *The manufacture of knowledge: An essay on the constructivist and contextual nature of science*. Elsevier.

Knuuttila, T. (2005). Models, representation, and mediation. *Philosophy of science*, *72*(5), 1260-1271.

Knuuttila, T. (2011). Modelling and representing: An artefactual approach to model-based representation. *Studies in History and Philosophy of Science Part A*, *42*(2), 262-271.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., ... & Wiedenbeck, S. (2011). The state of the art in end-user software engineering. ACM Computing Surveys (CSUR), 43(3), 1-44.

Koehler Leman, J., Weitzner, B. D., Renfrew, P. D., Lewis, S. M., Moretti, R., Watkins, A. M., ... & Bonneau, R. (2020). Better together: Elements of successful scientific software development in a distributed collaborative community. PLoS computational biology, 16(5), e1007507.

Kohler, R. E. (1991). Systems of production: Drosophila, Neurospora, and biochemical genetics. *Historical Studies in the Physical and Biological Sciences*, *22*(1), 87-130.

Kohler, R. E. (1994). Lords of the fly: Drosophila genetics and the experimental life. University of Chicago Press.

Kongsvik, T., Haavik, T., Bye, R., & Almklov, P. (2020). Re-boxing seamanship: From individual to systemic capabilities. *Safety science*, *130*, 104871.

Kuhn, T. S. (1962). The structure of scientific revolutions (Vol. 962). Chicago: University of Chicago press.

Lakatos, I. Worral, J. (ed.), Currie, G. (ed.) (1989). The methodology of scientific research programmes. Cambridge University Press.

Latour, B. (1983). Give me a laboratory and I will raise the world. *Science observed: Perspectives on the social study of science*, 141-170.

Latour, B. (1987). *Science in action: How to follow scientists and engineers through society*. Harvard university press.

Latour, B. (1992) 'Where are the missing masses? The sociology of a few mundane artifacts', in Bijker, W. E. and Law, J. (eds) Shaping Technology/Building Society: Studies in Sociotechnical Change, Cambridge, MA, MIT Press, pp. 225-58.

Latour, B. (2005). *Reassembling the social: An introduction to actor-network-theory*. Oxford university press.

Latour, B. & Woolgar, S. (1979). Laboratory life: The construction of scientific facts.

Laudan, L. (1977). Progress and its problems: Towards a theory of scientific growth (Vol. 282). Univ of California Press.

Law, J. (1987): Technology, Closure and Heterogeneous Engineering: The Case of the Portuguese Expansion. In W. Bijker, T. Pinch and T.P. Hughes (eds.): The Social Construction of Technological Systems. Cambridge, MA: MIT Press, pp. 111–134.

Lee, C. P. (2007). Boundary negotiating artifacts: Unbinding the routine of boundary objects and embracing chaos in collaborative work. *Computer Supported Cooperative Work (CSCW)*, *16*, 307-339.

Lee, C. P., Dourish, P., & Mark, G. (2006, November). The human infrastructure of cyberinfrastructure. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (pp. 483-492).

Lee, C. P., & Paine, D. (2015). From The Matrix to a Model of Coordinated Action (MoCA): A Conceptual Framework of and.

Leonardi, P. M. (2010). Digital materiality? How artifacts without matter, matter. *First monday*.

Leonardi, P. M. (2012). *Car crashes without cars: Lessons about simulation technology and organizational change from automotive design*. MIT Press.

Leonelli, S., & Ankeny, R. A. (2015). Repertoires: How to transform a project into a research community. *BioScience*, *65*(7), 701-708.

Lichtenstein, B. M. B. (2000). Generative knowledge and self-organized learning: reflecting on Don Schön's research. *Journal of Management Inquiry*, 9(1), 47-54.

Lyle, P., Korsgaard, H., & Bødker, S. (2020, October). What's in an ecology? A review of artifact, communicative, device and information ecologies. In *Proceedings of the 11th*

Nordic Conference on Human-Computer Interaction: Shaping Experiences, Shaping Society (pp. 1-14).

Lynch, M. (2002). Protocols, practices, and the reproduction of technique in molecular biology. *The British journal of sociology*, *53*(2), 203-220.

Lynch, M., & Edgerton Jr, S. Y. (1987). Aesthetics and digital image processing: Representational craft in contemporary astronomy. *The Sociological Review*, *35*(1), 184-220.

Mackenzie, A. (2006). *Cutting code: Software and sociality* (Vol. 30). Peter Lang.

MacKenzie, D. (2003). An equation and its worlds: Bricolage, exemplars, disunity and performativity in financial economics. *Social studies of science*, *33*(6), 831-868.

March, J. G., and H. A. Simon (1958). Organizations. New York: Wiley

Marshall, James; Robert R Downs, and Shahin Samadi (2010) Relevance of software reuse in building advanced scientific data processing systems. Earth Science Informatics 3, 1 (2010), 95–100.

McCray, W. P. (2006). *Giant telescopes: Astronomical ambition and the promise of technology*. Harvard University Press.

McDowell, J. C. (2020). The low earth orbit satellite population and impacts of the SpaceX Starlink constellation. *The Astrophysical Journal Letters*, *892*(2), L36.

McLaughlin, J., & Webster, A. (1998). Rationalising knowledge: IT systems, professional identities and power. *The Sociological Review*, *46*(4), 781-802.

Merton, R. K. (1938). Science, technology and society in seventeenth century England. *Osiris*, *4*, 360-632.

Morris, C., & Segal, J. (2009, December). Some challenges facing scientific software developers: The case of molecular biology. In *2009 Fifth IEEE International Conference on e-Science* (pp. 216-222). IEEE.

Middleton, J., Murphy-Hill, E., & Stolee, K. T. (2020). Data analysts and their software practices: a profile of the sabermetrics community and beyond. *Proceedings of the ACM on Human-Computer Interaction*, *4*(CSCW1), 1-27.

Mills, C. W. (2000). *The sociological imagination*. Oxford University Press.

Monteiro, E. (2022). *Digital oil: Machineries of knowing*. MIT Press.

Nardi, B. A. (1993). A small matter of programming: perspectives on end user computing. MIT press.

Neang, A. B., Sutherland, W., Ribes, D., & Lee, C. P. (2023). Organizing oceanographic infrastructure: the work of making a software pipeline repurposable. *Proceedings of the ACM on Human-Computer Interaction*, *7*(CSCW1), 1-18.

Nelson, R. R. and S. G. Winter (1982), An Evolutionary Theory of Economic Change, Cambridge, MA: Belknap Press.

Nersessian, N. J. (1988, January). Reasoning from imagery and analogy in scientific concept formation. In *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association* (Vol. 1988, No. 1, pp. 41-47). Cambridge University Press.

O'Brien, G. (2025, April). How Scientists Use Large Language Models to Program. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems* (pp. 1-16).

Obstfeld, D. (2020). *Getting new things done: Networks, brokerage, and the assembly of innovative action*. Stanford University Press.

Olson, G. M., Herbsleb, J. D., & Reuter, H. H. (1994). Characterizing the sequential structure of interactive behaviors through statistical and grammatical techniques. *Human–Computer Interaction*, *9*(3-4), 427-472.

Østerlie, T., & Monteiro, E. (2020). Digital sand: The becoming of digital representations. *Information and Organization*, *30*(1), 100275.

Paine, D. (2016). Software and Space: Investigating How a Cosmology Research Group Enacts Infrastructure by Producing Software (Doctoral dissertation).

Paine, D., & Lee, C. P. (2014, October). Producing data, producing software: Developing a radio astronomy research infrastructure. In 2014 IEEE 10th International Conference on e-Science (Vol. 1, pp. 231-238). IEEE.

Paine, D., & Lee, C. P. (2017). " Who Has Plots?" Contextualizing Scientific Software, Practice, and Visualizations. *Proceedings of the ACM on human-computer interaction*, *1*(CSCW), 1-21.

Parmiggiani, E. (2017). This is not a fish: on the scale and politics of infrastructure design studies. *Computer Supported Cooperative Work (CSCW)*, *26*(1), 205-243.

Partridge, R. B. (2019) The cosmic microwave background: from discovery to precision cosmology. In Kragh, H., & Longair, M. (Eds.). *The Oxford Handbook of the History of Modern Cosmology*. Oxford University Press.

Passi, S., & Barocas, S. (2019, January). Problem formulation and fairness. In *Proceedings of the conference on fairness, accountability, and transparency* (pp. 39-48).

Passi, S., & Sengers, P. (2020). Making data science systems work. *Big data & society*, *7*(2), 2053951720939605.

Paavola, S., & Miettinen, R. (2019). Dynamics of design collaboration: BIM models as intermediary digital objects. *Computer supported cooperative work (CSCW)*, 28, 1-23.

Pedersen, K. O. (1976). The development of Svedberg's ultracentrifuge. *Biophysical Chemistry*, *5*(1-2), 3-18.

Pentland, B. T., & Rueter, H. H. (1994). Organizational routines as grammars of action. *Administrative science quarterly*, 484-510.

Peterson, D., & Panofsky, A. (2021). Arguments against efficiency in science. *Social Science Information*, *60*(3), 350-355.

Pham, V. T. T., & Kelleher, C. (2025). Code histories: Documenting development by recording code influences and changes in code. *Journal of Computer Languages*, *82*, 101313.

Pickering, A. (Ed.). (2010). *Science as practice and culture*. University of Chicago press.

Pickering, A. (1992). From science as knowledge to science as practice. *Science as practice and culture*, *4*.

Pinch, T. (1993). " Testing-One, Two, Three... Testing!": Toward a Sociology of Testing. *Science, Technology, & Human Values*, *18*(1), 25-41.

Pinheiro, F. A., & Goguen, J. A. (2002). An object-oriented tool for tracing requirements. *IEEE software*, *13*(2), 52-64.

Pink, S., Ardèvol, E., & Lanzeni, D. (2020). Digital materiality. In *Digital materialities* (pp. 1-26). Routledge.

Polanyi, M. (2013 [1951]). The logic of liberty: Reflections and rejoinders. Routledge.

Polanyi, M. (1945). The autonomy of science. The Scientific Monthly, 60 (2), 141-150.

Popper K (1957) The Poverty of Historicism. London: Routledge.

Popper, K. R. (1959) The Logic of Scientific Discovery. London: Hutchinson and Company.

Popper, K. (2014). *Conjectures and refutations: The growth of scientific knowledge*. routledge.

Ram, Karthik, Carl Boettiger, Scott Chamberlain, Noam Ross, Maëlle Salmon, and Stefanie Butland. 2018. A community of practice around peer review for long-term research software sustainability. Computing in Science & Engineering 21, 2 (2018), 59–65.

Rheinberger, H. J. (1992a). Experiment, difference, and writing: I. Tracing protein synthesis. *Studies in History and Philosophy of Science Part A*, *23*(2), 305-331.

Rheinberger, H. J. (1992b). Experiment, difference, and writing: II. The laboratory production of transfer RNA. *Studies in History and Philosophy of Science Part A*, *23*(3), 389-422.

Rheinberger, H. J. (1997). Toward a history of epistemic things: Synthesizing proteins in the test tube. Stanford University Press.

Rheinberger, H. J. (2005). Gaston Bachelard and the notion of “phenomenotechnique”. *Perspectives on Science*, *13*(3), 313-328.

Rheinberger, H. J. (2016). Science and Experiment. *Scientific Knowledge and the Transgression of Boundaries*, 23-33.

Rheinberger, H. J. (2020). *An epistemology of the concrete: Twentieth-century histories of life*. Duke University Press.

Ribes, D. (2014, February). Ethnography of scaling, or, how to fit a national research infrastructure in the room. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing* (pp. 158-170).

Ribes, D., & Finholt, T. A. (2008, November). Representing community: knowing users in the face of changing constituencies. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work* (pp. 107-116).

Ribes, D., & Finholt, T. A. (2009). The long now of infrastructure: Articulating tensions in development.

Ribes, D., & Polk, J. B. (2015). Organizing for ontological change: The kernel of an AIDS research infrastructure. *Social Studies of Science*, *45*(2), 214-241.

Roberts RM (1989) Serendipity. Accidental discoveries in science. Wiley, New York

Rother, K., Potrzebowski, W., Puton, T., Rother, M., Wywial, E., & Bujnicki, J. M. (2012). A toolbox for developing bioinformatics software. *Briefings in bioinformatics*, *13*(2), 244-257.

Rule, A., Drosos, I., Tabard, A., & Hollan, J. D. (2018). Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proceedings of the ACM on Human-Computer Interaction*, *2*(CSCW), 1-12.

Rule, A., Tabard, A., & Hollan, J. D. (2018, April). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1-12).

Saldaña, J. (2021). The coding manual for qualitative researchers.

Sanchez, R., & Mahoney, J. T. (1996). Modularity, flexibility, and knowledge management in product and organization design. *Strategic management journal*, 17(S2), 63-76.

Schank, R. C., & Abelson, R. P. (2013). *Scripts, plans, goals, and understanding: An inquiry into human knowledge structures*. Psychology press.

Schmidt, K. (1994). Cooperative work and its articulation: requirements for computer support. *Le travail humain*, 345-366.

Schmidt, S. (2009). Shall we really do it again? The powerful concept of replication is neglected in the social sciences. *Review of general psychology*, *13*(2), 90-100.

Schmidt, K. (1997, November). Of maps and scripts—the status of formal constructs in cooperative work. In *Proceedings of the 1997 ACM International Conference on Supporting Group Work* (pp. 138-147).

Schmidt, K., & Bannon, L. (1992). Taking CSCW seriously: Supporting articulation work. *Computer supported cooperative work (CSCW)*, 1, 7-40.

Schmidt, K., & Simone, C. (1996). Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. Computer Supported Cooperative Work (CSCW), 5(2), 155-200.

Schmidt, K., & Wagner, I. (2002, June). Coordinative artifacts in architectural practice. In COOP (Vol. 2, pp. 257-274).

Schmidt, K., & Wagner, I. (2004). Ordering systems: Coordinative practices and artifacts in architectural design and planning. Computer Supported Cooperative Work (CSCW), 13, 349-408.

Schön, D. (1983). The reflective practitioner. New York: Basic Books.

Schön, D. A. (1992). The theory of inquiry: Dewey's legacy to education. *Curriculum inquiry*, 22(2), 119-139.

Scroggins, M., & Boscoe, B. M. (2020). Once FITS, always FITS? Astronomical infrastructure in transition. *IEEE Annals of the History of Computing*, *42*(2), 42-54.

Segal, J. (2007, September). Some problems of professional end user developers. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007) (pp. 111-118). IEEE.

Segal, J. (2008). Scientists and software engineers: A tale of two cultures. In: PPIG 2008: Proceedings of the 20th Annual Meeting of the Psychology of Programming Interest Group (Buckley, Jim; Rooksby, John and Bednarik, Roman eds.), Lancaster University, Lancaster, UK.

Segal, J. (2008). Models of scientific software development. SECSE 08, First International Workshop on Software Engineering in Computational Science and Engineering, 13 May 2008, Leipzig, Germany.

Segal, J. (2009). Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community, Computer Supported Cooperative Work 18 581–606.

Sellen, A. J., & Harper, R. H. (2003). *The myth of the paperless office*. MIT press.

Sepkoski, D. (2017). The Database before the Computer?. *Osiris*, *32*(1), 175-201.

Shapin, S. (1989). The invisible technician. *American scientist*, *77*(6), 554-563.

Shapin, S., & Schaffer, S. (2011). *Leviathan and the air-pump: Hobbes, Boyle, and the experimental life*. Princeton University Press.

Sims, Benjamin. 2022. Research software engineering: Professionalization, roles, and identity. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Simon, H. A. (1973). Does scientific discovery have a logic?. *Philosophy of science*, *40*(4), 471-480.

Sletholt, M. T., Hannay, J., Pfahl, D., Benestad, H. C., & Langtangen, H. P. (2011, May). A literature review of agile practices and their effects in scientific software development. In *Proceedings of the 4th international workshop on software engineering for computational science and engineering* (pp. 1-9).

Smith, A. M., Katz, D. S., & Niemeyer, K. E. (2016). Software citation principles. *PeerJ Computer Science*, *2*, e86.

Spencer, M. (2015). Brittleness and bureaucracy: software as a material for science. Perspectives on Science, 23(4), 466-484.

Star, S. L. (1985). Scientific work and uncertainty. Social studies of Science, 15(3), 391-427.

Star, S. L., & Bowker, G. C. (2006). How to infrastructure. *Handbook of new media: Social shaping and social consequences of ICTs*, 230-245.

Star, S. L., & Gerson, E. M. (1987). The management and dynamics of anomalies in scientific work. *Sociological Quarterly*, 28(2), 147-169.

Star, S. L., & Griesemer, J. R. (1989). Institutional ecology, ‘translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. Social studies of science, 19(3), 387-420.

Star, S. L., & Ruhleder, K. (1994, October). Steps towards an ecology of infrastructure: complex problems in design and access for large-scale collaborative systems. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work* (pp. 253-264).

Steinle, F. (1997). Entering new fields: Exploratory uses of experimentation. *Philosophy of science*, *64*(S4), S65-S74.

Stodden, V. (2020, June). Beyond open data: a model for linking digital artifacts to enable reproducibility of scientific claims. In *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems* (pp. 9-14).

Stodden, V., Krafczyk, M. S., & Bhaskar, A. (2018, June). Enabling the verification of computational results: An empirical evaluation of computational reproducibility. In *Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems* (pp. 1-5).

Stol, K.J., P. Ralph, and B. Fitzgerald, (2016) “Grounded theory in software engineering research: A critical review and guidelines,” in Proc. 38th Int. Conf. Softw. Eng., pp. 120–131.

Strasser, B. J. (2019). *Collecting experiments: Making big data biology*. University of Chicago Press.

Strauss, A. (1988). The articulation of project work: An organizational process. *Sociological Quarterly*, *29*(2), 163-178.

Stylos, J., & Myers, B. (2007, September). Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)* (pp. 50-60). IEEE.

Suchman, L. A. (1987). *Plans and situated actions: The problem of human-machine communication*. Cambridge university press.

Sullivan, W. T., & Bertotti, R. (1990). The entry of radio astronomy into cosmology: radio stars and Martin Ryle’s 2C survey. *Modern cosmology in retrospect*, 309-330.

Sutherland, W., Paine, D., & Lee, C. P. (2024). ‘The Cloud is Not Not IT’: Ecological Change in Research Computing in the Cloud. *Computer Supported Cooperative Work (CSCW)*, 1-33.

Tamborini, M. (2020). Technoscientific approaches to deep time. *Studies in History and Philosophy of Science Part A*, *79*, 57-67.

Tavory, I., & Timmermans, S. (2014). *Abductive analysis: Theorizing qualitative research*. University of Chicago Press.

Timmermans, S., & Tavory, I. (2012). Theory construction in qualitative research: From grounded theory to abductive analysis. *Sociological theory*, *30*(3), 167-186.

Trainer, E. H., Chaihirunkarn, C., Kalyanasundaram, A., & Herbsleb, J. D. (2015, February). From personal tool to community resource: What's the extra work and who will do it?. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing* (pp. 417-430).

Turk, M. J. (2013, July). Scaling a code in the human dimension. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery* (pp. 1-7).

URSSI (2021) US Research Software Sustainability Institute (URSSI). (2021). https://urssi.us/

Vaughan, D. (1992). Theory elaboration: The heuristics of case analysis. *What is a case*, *173202*.

Vertesi, J. (2020). Shaping science: Organizations, decisions, and culture on NASA's teams. In *Shaping Science*. University of Chicago Press.

Vinck, D. (2011). Taking intermediary objects and equipping work into account in the study of engineering practices. Engineering Studies, 3(1), 25-44.

Vinck, D. (2012). Accessing material culture by following intermediary objects. An ethnography of global landscapes and corridors, 89-108.

Wastell, D. (1999). The human dimension of the software process. *Software Process: Principles, Methodology, and Technology*, 165-199.

Weick, K. E. (1995). *Sensemaking in organizations* (Vol. 3, pp. 1-231). Thousand Oaks, CA: Sage publications.

Weick, K. E. (1998). Introductory essay—Improvisation as a mindset for organizational analysis. *Organization science*, *9*(5), 543-555.

Wells, D. C. (1985). NRAO's astronomical image processing system (AIPS). In *Data analysis in astronomy* (pp. 195-209). Boston, MA: Springer US.

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., ... & Wilson, P. (2014). Best practices for scientific computing. PLoS biology, 12(1), e1001745.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLoS computational biology*, 13(6), e1005510.

Wilson, G. (2006a). Software carpentry: getting scientists to write better code by making them more productive. Computing in science & engineering, 8(6), 66-69.

Wilson, G. (2006b). Where's the real bottleneck in scientific computing? *American Scientist*, 94(1), 5.

Wittenhagen, M., Cherek, C., & Borchers, J. (2016, May). Chronicler: Interactive exploration of source code history. In *Proceedings of the 2016 CHI conference on human factors in computing systems* (pp. 3522-3532).

Woolgar, S. (1990). Configuring the user: the case of usability trials. *The Sociological Review*, *38*(1), 58-99.

Woolgar, S., & Grint, K. (1991). Computers and the transformation of social analysis. *Science, Technology, & Human Values*, *16*(3), 368-378.

Woolgar, S. W. (1976). Writing an intellectual history of scientific development: The use of discovery accounts. *Social Studies of Science*, *6*(3-4), 395-422.

Ziewitz, M. (2017). A not quite random walk: Experimenting with the ethnomethods of the algorithm. *Big Data & Society*, *4*(2), 2053951717738105.

Appendix

Appendix A: Codebook

Top level code	Code	Description
Exploratory testing	Elaborating the phenomenon / Transforming the problem	Elaborating the phenomena describes the iterative process of establishing new characteristics of a poorly understood phenomenon. This was a state of interaction between the researcher and an object of investigation and the researcher might remain in this progressive but indeterminate state for great lengths of time.
	Seeing what happened	An essential characteristic of exploratory work is that the group must closely examine what happened post hoc. This is a kind of retrospection that guides future action. Activities such as the data rampage are facilitated, intentional processes in this direction.
	Anomaly	Anomalies are disjunctures in the working order of an instrumental system. They are also points for potential expansion of the system. Often, they were the centerpiece of investigations in the Radio Group, providing warrant for further investigation as well as incomplete but engageable shape, which could help shape future action.
	Testing rate	Because the Radio Group had little clarity about the future directions of their work, they placed a premium on being able to try things out and see what happens. In particular this was a matter of having a rapid testing rate. In the context of high performance computing this notion of rate can be seen as an alternative to being able to run large, monolithic tests that require lots of computing power. The key aspect is the iteration.
	Reproduction	The Radio Group iterates on tests and on phenomena. They do this by drawing on established code and routines for testing and plotting phenomena. This process is one of reproduction, but in a way that leverages routine action in an intentionally generative way.
Collaboration software	Planning development	In the context of collaboration software the Radio Group actively planned development of the tools well in advance. This was done primarily through the collection of "issues" into "milestones." This is a notable difference from exploratory work, which usually had a central anomaly and a larger end goal, but little clear view of what the software ought to be doing.

	Reproblematizing established software	Established and well-tested software sometimes became problematic once again. This could happen when robust code was used in new circumstances, especially with new datasets, but it also happened whenever the software now needed to accommodate a new stakeholder. New stakeholders brought with them new goals and intentionalities, which meant the software was now being used in engaging new problems.
	Contributing	Collaboration software was built between multiple contributors, and contributing to such a software project involved programming under the view of others. It also entailed working through certain kinds of contribution requirements and standards that had been set up around repositories.
	A lot of people in the loop	"A lot of people in the loop" is an in vivo code that I adapted from one of my interlocutors in order to make sense of things that many people said about contributing to collaboration software. That is that in doing software production, building collaboration software, one has to engage and negotiate with others, making their code accommodate the needs of other users and not obstruct other purposes of the code. A lot of people in the loop capture the collaborative framing of production code but also the coordinative overheads that researchers associated with this kind of software.
	Software reliability	The reliability of research software is a complex concept. It involves understanding of consistency in processing, a kind of reliability embodied in unit testing and continuous integration practices. The notion of consistent processing had replicative benefits in that the researcher can get a phenomenon again or in a similar way to a previous test. However, it also meant that they could "have" it the same way that a collaborator had it, implying consistency across researchers. Reliability was not just a matter of conformity to unit tests, but also the situation of development and maintenance on the software project, such that a researcher knew that even if software broke it could be investigated and fixed by dedicated developers. In this sense it was the reliability of a network of actors.
Code growth	Developing capacities	What the research software system develops over time is new capacities for experiencing and interpreting the world. These are performative interactional potentialities, which take the instrument and the phenomenon together.
	Knobs	Research software used for iterative reproduction of anomalies were designed with knobs or parameters that could be used to slightly adjust or reconfigure tests, allowing for the production of phenomena in ways that were similar but also different from prior tests. These knobs accumulated over time and were hard

		to keep track of, in part because it was not clear when one would need them again and whether they could be removed or redesigned.
	Graduating code	Graduating code was a movement between distinct ways of working with software. It first relies on a process of selecting functionality that is worth turning into collaboration software (see defining collaboration functionality). It then involves rewriting, redesigning, documenting, testing, and contributing code in order to turn it into collaboration software. This is the transition to a mode of software production and it was a negotiated thing when and whether to graduate code.
	Residual code	Researchers wrote a great deal of code in the course of investigating anomalies and research problems. Residual code refers to a situation where this code inhabits a grey area where it is not obviously useful enough to warrant graduation and frequent reuse, but it also could be useful again or might be needed as a record of tests performed.
	Defining collaboration functionality	Defining collaboration functionality was the process of selecting functionality that was worth graduating to collaboration code. The primary criterion for this was whether something that would be reused frequently would be reused by others. However, in the course of exploratory work sometimes functionality was not understood to be collaboration functionality until it began to be reused by people.
Instrumental opacity	Silent bugs	One of the primary threats to the working of an instrumental system was the possibility of a bug being introduced and the researcher not realizing that it had happened. This was a threat to potential outputs of the Radio Group's analysis software, but perhaps more importantly it could lead to misleading or confounding results in the exploratory process, precisely when the researchers were trying to achieve clarity about phenomena.
	Mnemonic artifacts	Mnemonic artifacts are artifacts which enable the researcher to forget the details of the software, or simply not have them in mind, because they make parts of a large, opaque system available when they become relevant to ongoing work. They do not themselves remember or encode past action, they simply alert and direct the attention of the developer.
	Lab memory	Members of the Radio Group would contact or confer with others who wrote a particular piece of code or who were there when decisions about it were being made. This was in part an alternative to and in complement to mnemonic artifacts as a way of maintaining understanding of a large complex software

		system.
	Memos	Members of the radio group would write memos about the design of particular parts of their software system in order to leave a record of why certain decisions were made and what the rationale for the design was at a particular time. This was a way of maintaining documentation external to the code itself, but they were often kept within the software repositories to which they were relevant. It was done in particular for complex pieces of code or for operations which had rationales grounded deeply in physics.
Championing	Software epithets	Members of the Radio Group and their collaborators had a variety of terms and descriptions of software which emphasized different materialities and moral valences that they embodied. These terms and nicknames were ways of differentiating a general category of material for their work into different kinds of entities with different uses or challenges.
	Exemplar	Software packages that were developed using software engineering practices such as unit tests and continuous integration could become exemplars of how to use those practices and of what 'good', reliable research software should look like. Exemplars served as demonstrations of what these practices looked like and how to implement them.
	Sharing coding practices	In the course of working together on software packages or in helping others contribute to software, researchers would share tools and techniques for software development. This was a way that such practices moved throughout the Radio Group's larger collaboration. Through this process the repository became a place or venue for the promotion of new practice.
	Disciplining software practice	Researchers understood learning and implementing software engineering practices as a matter of discipline: something that they ought to do but which was difficult, labor intensive, or a nuisance. Adhering to this discipline became the basis for both rigor and pride in software-related work.
	Shepherding contributions	Facilitating and teaching others in new software engineering practices occurred in part through the activities of soliciting, guiding, reviewing, and structuring (providing templates for) contributions to repositories. Encouraging this activity and making it easier encouraged others to use these practices, but it also created the occasion for people to learn and implement these practices.

Appendix B: Glossary of terms

Term	Description
Array	The interferometers that the Radio Group uses collect data through a large array of radio antennae, which operate together as a single instrument. The “array” refers to a specific set of antennae operating together in this way, and by extension to the hardware used on site to capture and record data from the antennae.
Best practices	A number of papers have presented best practices for software engineering. These practices are intended to reflect the experience of the authors as software engineers and are meant to provide simple and minimal rules that researchers can follow to improve the way they work with software.
Commissioning	A process that an interferometer goes through early on to establish that it is in working order and can collect data. It involves working out major bugs and breakdowns in both the hardware and software.
Continuous integration	Continuous integration describes a development practice and a suite of tools to support that practice. The practice itself involves making frequent code contributions and testing new code early in the development process. The suite of development tools include systems for automated testing of code based on unit tests that have been written. These can be run by Github automatically whenever a new pull request is created on a repository. These tools are often referred to themselves as “CI tools.”
Cosmic Dawn Array (CDA)	One of the interferometer projects the Radio Group works with and their most current project. Involves a number of research groups at different universities, who work on designing, operating, and analyzing data from a single hardware instrument.
Flagging	A process of marking parts of a data set with “flags” indicating that the data is contaminated with radio frequency interference or has some other quality that the researcher should be aware of.
High Altitude LOw Frequency Array (HALO)	An interferometer facility that collects data for a wide variety of science cases. While it has no particular connection to the science of reionization cosmology, certain members of the facility committed to using pycosmo as a stable software package for manipulating and working with their data.
Interferometer	A type of instrument used in radio astronomy which comprises a large array of antennae operating together as a single detector. Interferometers are used elsewhere for other purposes, but they are a common instrument in radio astronomy and the center of the Radio Group’s work.
Issue	The code host website Github provides a ticketing system whereby

	problems or bugs with existing code can be logged. In Github's terminology a record of a bug or requested change to the code (or some other aspect of the repository) is referred to as an "issue." Developers need not use this system, but it is common practice to write issues in order to maintain a backlog of things to work on, or requests or bug reports from users. Issues can be linked to pull requests (described below) and are "closed" if a pull request addresses them.
Oracle	A term used in software engineering to describe a gold standard that defines the output of a piece of software is correct given a particular input. This term is often used in the context of testing and is sometimes referred to as a testing oracle.
Pull request	A standard operation that can be performed on the software repository host Github. A pull request puts forward a new version of the codebase with a person's changes included. A webpage is created where the requester and developers on the project can see specific points of change in the code and they can discuss the changes. This is often the point at which tests are run to indicate whether the new changes break other existing parts of the code. If the changes are approved they are applied to the codebase and the pull request is "closed." Other code hosting systems may not use this term "pull request", but they facilitate a similar operation.
Pycosmo	A software package developed by researchers from a number of different labs and universities, most of whom work in the field of reionization cosmology. Pycosmo was intended as a data format converter and data manipulation tool ("glue code") that could be used potentially with many different kinds of interferometric data. The package became an exemplar of good development practice for the Radio Group and their collaborators in the CDA.
Radio Frequency Interference	Low frequency emissions of any type that might be collected by an interferometer or other radio telescope and interfere or obstruct the researcher's ability to see what they are looking for. In most cases radio frequency interference refers to human-made emissions from TV broadcasts, satellites, and other sources. A large part of the Radio Group's work is detecting and flagging parts of the data that have this interference, such that they will not be used in analysis.
Requirements	Requirements are specifications of what software should do and under what constraints it should do it. It includes an array of different kinds of specifications, from specific outputs to descriptions of certain priorities that the software must work with, such as not using much memory or being highly optimized, or working on particular operating systems.
Widefield Radio Telescope (WRT)	One of the interferometer projects the Radio Group works with. In contrast to the CDA, the WRT is an interferometer that is used for a number of different kinds of science beyond reionization cosmology. There are, however, a number of other research groups in the field that the Radio Group has interacted with through the WRT, and a large part

	of their data analysis software was developed initially to process data from the WRT.