

**Regenerative Coordination:
Keeping a Live Algorithmic Service Growing by Perpetuating Disruptions**

By

Alan Zhang

B.S. Neuroscience
Washington University in St. Louis, 2013

M.S. Customer Analytics
Washington University in St. Louis, 2017

SUBMITTED TO THE DEPARTMENT OF MANAGEMENT IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN MANAGEMENT RESEARCH
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2021

©2021 Massachusetts Institute of Technology. All rights reserved.

Signature of Author: _____
Department of Management
August 6, 2021

Certified by: _____
Wanda Orlikowski
Alfred P. Sloan Professor of Management
Professor of Information Technologies and Organization Studies
Thesis Supervisor

Accepted by: _____
Catherine Tucker
Sloan Distinguished Professor of Management
Professor of Marketing
Faculty Chair, MIT Sloan PhD Program

**Regenerative Coordination:
Keeping a Live Algorithmic Service Growing by Perpetuating Disruptions**

By

Alan Zhang

Submitted to the Department of Management
on August 6, 2021 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Management Research

ABSTRACT

Many organizations today, in their move to online platforms, seek to provide a ‘live’ service—a digital service capable of updating automatically, offering users continual improvements in content and functionality. I examined the work required to keep such a service going, and found developers struggling to coordinate their work in the face of heterogeneous, concurrent, and indefinite updates. My 15-month field study of an agricultural technology company explores how its members, along with its algorithms, were able to sustain a live imagery-analytics service, despite frequent, unexpected, and disruptive updates. Existing literature shows that sense-making and provisional settlements can be critical for coordinating distributed and dynamic work, but takes a perspective which centers on human actors. Taking a broader perspective, I suggest that the algorithms working in a live service may do neither. I found that algorithms frequently updated operations with changes unanticipated by members, and in so doing, disrupted operations that members had previously settled and, until then, considered usable. Members responded to these updates with further updates required to regenerate the service, but with each update new disruptions emerged. Drawing on these findings, I develop the notion of regenerative coordination that identifies the specific coordination practices that regenerate a live service through updates, thus keeping the service viable and valuable to users. Doing so however, perpetuates disruptions. This paradoxical outcome is the inadvertent result of a process that keeps a service live and growing. I end with contributions to coordination research.

Thesis Supervisor: Wanda Orlikowski

Title: Alfred P. Sloan Professor of Management and Professor of Information Technologies and Organization Studies

Introduction

Modern algorithms enable a key offering of digital platforms: a “live” service. A live service can update indefinitely from algorithms of the installed platform, like Spotify’s playlists-as-a-service (Nieborg & Poell, 2018) and Tesla’s operating system-as-a-service (Acker & Beaton, 2016). For platform users (including end-users, content contributors, and developers), algorithms stream new features, delivering live updates regularly without users explicitly requesting or installing them. Users’ interactions with a service, in providing algorithms with real-time input on the cloud, contribute to updates that affect other users without their awareness. This dynamic and distributed form of delivering a live service falls on workers to sustain. Whereas traditionally, workers can say when done is done, once a service becomes live it is open to the indefinite inputs of many users. Workers thus get caught in an endless stream of updates, making coordination difficult. The coordination literature offers a number of insights about coordinating distributed and dynamic work, including dedicating time to deliberate (Bruns, 2013), leveraging deadlines to reach provisional settlements (Beane & Orlikowski, 2015), and sensing local cues to adapt appropriately (LeBaron et al., 2016). However, these practices are unlikely to be feasible in a live algorithm-based service where workers lack dedicated time, opportunities to settle ends, and locally visible cues. I present a new model of coordination for the continual work of developing a live service.

Live services are proliferating (Cusumano, 2008; Manikas, 2016) in part because many organizations have shifted away from discrete deliverables and packaged products, to focus more on continuance of digital services (Bhattacharjee, 2001). Powered largely by platform models (Stark & Pais, 2020) and cloud-based algorithms (Morreale & Eriksson, 2020), live services have already transformed the production of games (Švelch, 2019), news (Nieborg & Poell, 2018),

music (Arditi, 2018), and mobility (Acker & Beaton, 2016). A live service benefits from an open-ended timeframe for production, enabling developments to be meted out in anytime updates, even after release of the service. Departing from the conventional sequence—develop, finalize, deliver, then use (Bechky & Chung, 2018; Whyte et al., 2016)—a live service is a perpetual work-in-progress, developing concurrently in use. And because updates originate from heterogenous users — including data providers, open code communities, vendor services, client end-users, and the service’s own developers — each following their own objectives, updates are seldom aligned. When disruptions to service arise as a result, workers must address them absent the development time periods, settlement processes, and embodied experiences associated with known forms of coordination.

Prior literature is largely based on settings where workers can expect certain outcomes by a certain deadline. There is often a defined period for development whose end is marked by usable outputs, like project deliverables (Kellogg et al., 2006) and manufactured products (Lifshitz-Assaf et al., 2020). In settings where use and development unfold together, the end is marked by some event, like the anticipated discharging of a patient (Faraj & Xiao, 2006) or extinguishing of a fire (D. Geiger et al., 2020). Coordinating such work is difficult because conditions can be highly dynamic from start to end, but there typically is an end. Conditions are more complicated with live, algorithm-based services that exist through indefinite updates. The existing literature offers few insights on coordinating work amid updating algorithms.

To address this gap, I conducted a 15-month field study of an analytics service in a market-leading agricultural technology company. My research question has two parts: how is work coordinated in conditions of heterogenous, concurrent, and indefinite updating? And what consequences result from coordinating this way? To answer these questions, I studied a live,

algorithm-based service that generated analytics based on daily-updating of planetary-scale satellite imagery for over five billion acres of global cropland. Updates spanned more than twenty-five live data sources, hundreds of software services, and thousands of cloud computers. By studying the members who kept developing the live service, I identified a novel form of coordination—*regenerative coordination*—which may be increasingly prevalent as cloud services are becoming widespread.

My research explicates how regenerative coordination can support work in conditions of heterogeneous, concurrent, and indefinite updating. The aim of regenerative coordination is to keep current operations consistent and continuous despite continual updates. It does this by managing updates in real-time, and ensuring that the disruptions arising unexpectedly from updates do not cause a live service to cease or fail, but regenerate the service instead. I distinguish three regenerative practices, which entail workers incorporating improvements, detecting disruptions, and restoring functionality. These practices sustain a live service not merely by averting failure, but as I show, by perpetuating growth through disruptions. I find that disruptions caused by updates fuel further updates, and explicate how the urgency of these disruptions help keep a service live and growing. The consequence is a seemingly endless push to keep developing the service, generating further updates and further disruptions. The commitment to keep a live service ongoingly viable and valuable to users requires perpetuating disruptions, a paradoxical outcome that arises unavoidably and inadvertently from the work of keeping a service going.

I start with a review of the coordination literature, organized around three challenges known to hinder coordination, and describe the approaches scholars have proposed for overcoming them. I then consider the challenge of coordinating a live, algorithm-based service,

and explain how existing forms of coordination are not feasible in these conditions. In my methods, I describe the particular kind of analytics service I studied, and characterize its application in agriculture. Details about the data gathering and analytical processes for my field study are then provided. Drawing on this evidence, I identify three coordination practices which, by managing updates and disruptions, are able to continually regenerate the live, algorithm-based service. I theorize the notion of regenerative coordination, and show how this way of working through updates inadvertently perpetuates disruptions—with updates causing disruptions warranting more updates. I conclude with contributions to the coordination literature.

Literature Review

The literature defines coordination as the integration of work under task interdependence and uncertainty (Faraj & Xiao, 2006). Scholars have found that coordination can be strained across physical (Hinds & Bailey, 2003), occupational (Valentine & Edmondson, 2015), and temporal boundaries (Beane & Orlikowski, 2015), and complicated by changing work conditions (Bechky, 2009), objectives (Kellogg et al., 2006), and routines (Stephens, 2020). I organize this literature around three challenges that boundaries and change have been identified to pose for coordination: connecting distant resources, aligning objectives, and adapting to local contexts. After examining these challenges, I describe a fourth challenge which I see arising from the role of algorithms in a live service (see **Table 1** for review of coordination challenges).

Challenge of Connecting Distant Resources

Members do not always have what they need, where and when they need it. Their work may then falter when critical resources—knowledge and information in particular—are not accessible. Making distant resources available and accessible thus facilitates coordination. The challenge however, is that specialized knowledge is difficult to mobilize. To this end, some

studies present structural mechanisms for information flow and knowledge translation across boundaries (Kellogg et al., 2006). Others suggest practices for information transfer when local understandings change (Feldman & Rafaeli, 2002), which involve organizational designs (Parnas et al., 1985), positions (Dahlander & O’Mahony, 2011), teams (Faraj & Sproull, 2000), and tools (Malone & Crowston, 1994). Scholars have also examined the use of relational toolkits (DiBenigno & Kellogg, 2014), protocols (Whyte et al., 2016), and representations (W. C. Barley, 2015), highlighting their informal (Brosius et al., 2017) and emergent qualities.

Coordination Challenges	Benefits for the work	Source of boundaries and change	Issue arising from boundaries	Issue arising from changes	Activities to coordinate	Aim of coordination	Relevant studies
Connecting distant resources	Access to specialized resources	Distributed information and activities	Relevant information does not flow or translate	Locally available information may be inappropriate or irrelevant	Create structures and pathways for information flow	Synchronicity, and connectivity	Terwiesch et al. (2002); Kellogg et al. (2006)
Aligning objectives	Aligned task progress	Decoupled, concurrent progress	Overall progress is obscured, and may be unknowingly misaligned	Local activities may not be in accord with one another	Foster periods for deliberation and settlement across parallel activities	Compatibility and usability	Harrison & Rouse (2014); Bruns (2013)
Adapting to local contexts	Flexibly appropriate action	Inconstant, emergent situations	Access to crucial cues are delayed or misunderstood	Existing plans, scripts, and routines may fail to cover exigent needs	Expose members to cues so they can sense-make and improvise autonomously	Immediacy and adaptability	Faraj & Xiao, 2006; Geiger et al., 2020
Sustaining a live service	Rapid and continual development	Heterogeneous, concurrent, indefinite updates	Distant updates can alter local configurations with minimal notifications	Unexpected changes create urgent disruptions in settled work	Enroll digital actors in improving, detecting, and restoring work	Consistency and continuity	Current study

Table 1: Coordination challenges

Two studies convey this challenge well. Kellogg et al. (2006), describing how members connect through distributed information, introduce the notion of “trading zones.” They find that through the ongoing enactment of trading zones, individual members can “draw on diverse emerging interests and information to accomplish their complex, dynamic, and heterogeneous project-based work” (42) across community boundaries. Specifically, members in the trading

zone coordinate by making commitments visible, rendering work legible, and assembling work through ongoing alignment. In another study, Terwiesch et al. (2002) examine distributed activities, for which delays can arise when downstream activities wait on information upstream to become available. To resolve this, Terwiesch et al. (2002) recommend making activities modular and concurrent (Sanchez & Mahoney, 1996), and amenable to “iterative coordination,” so that work can proceed without full information, incorporating relevant information iteratively as it becomes available.

Challenge of Aligning Objectives

Members can have access to requisite resources and yet not be coordinated, if objectives are not aligned. Occupied by different tasks, members with different expertise and experiences may not realize the bigger picture to which they contribute. Misaligned objectives can lead to conflicting and discrepant work. The challenge arises, in part, because the boundaries distancing members from one another can obscure objectives. Valentine & Edmondson (2015) find that the use of team scaffolds can help to keep objectives aligned when stable team composition is lacking. The challenge of aligning objectives may also arise because objectives themselves change. Bechky (2006) finds that, to deal with change, a combination of structural elements and role enactments can help people form appropriate expectations. These approaches seek to improve the compatibility and usability of coordinated work by aligning objectives.

Other approaches for aligning objectives highlight the importance of members noticing whether they are aligned. One approach is to smooth over disagreements, to systematize work and keep it more focused on common ground than on differences. So long as outcomes remain usable, members may not be aware of latent disagreements and misunderstandings. This can result from strategic ambiguity (W. C. Barley et al., 2012), plug-and-play teaming (Faraj & Xiao,

2006), joint assessments (Bruns, 2013), abstraction of details (Fayard & Metiu, 2014), and provisional task orders (Kremser & Blagoev, 2020). Another approach, in contrast, is to highlight disagreements, revealing discrepancies and spotlighting opportunities for realigning. This approach involves generating disputes and exposing deviations (LeBaron et al., 2016), fragmentations (Stephens, 2020), missing processes (Jarzabkowski et al., 2012), nuanced distinctions (Mengis et al., 2018; Tuertscher et al., 2014), discontinuities (Harrison & Rouse, 2014), puzzles (Beane & Orlikowski, 2015), and failing trajectories (Faraj & Xiao, 2006). For example, Bruns (2013), studying long-term cancer research projects, found scientists embracing unfamiliar suggestions by counter-projecting and then executing faithfully on joint assessments, which “alerted scientists to changes they needed to make... so that their contribution[s] would correspond” (p. 73).

Speed is a limiting factor for these two approaches. They are not immediate, taking time to unfold, maybe days or longer. People finding common ground at one point of time may later generate disputes, and vice versa. According to the literature, this process of deviating and aligning unfolds sequentially, either linearly (Beane & Orlikowski, 2015; Bechky & Okhuysen, 2011; Faraj & Xiao, 2006; LeBaron et al., 2016; Lifshitz-Assaf et al., 2020; Tuertscher et al., 2014, Lifshitz-Assaf et al. 2020), or cyclically (Bruns, 2013; Harrison & Rouse, 2014, 2014; Jarzabkowski et al., 2012; Kremser & Blagoev, 2020; Stephens, 2020), but not concurrently. A clear example of this sequential temporality is Harrison & Rouse’s (2014) notion of “elastic coordination,” which depicts workers shifting between periods of systematizing—when members “de-integrate” work so as not to butt heads—and periods of problematizing—when members rejoin and iron out differences. Similarly, Faraj & Xiao’s (2006) “dual nature” portrayal of coordination, depicts “habitual trajectories” of work (i.e., normal information use) as coming

either before or after “problematic trajectories” (i.e., contentious information use), but not at the same time. Regardless of order, these activities help create the “integrative conditions”

(Okhuysen & Bechky, 2009) in which distributed actors can align objectives.

Challenge of Adapting to Local Contexts

To coordinate effectively, members need not only to align on general objectives but also to adapt to the particular situations at hand. This challenge is salient when work recurs in changing contexts. In each occurrence, local adaptations are crucial. It is not enough for members to know standard rules, routines, or protocols, they also have to act appropriately in specific circumstances. This can be difficult if boundaries obscure local conditions, or if conditions change too quickly. Geiger et al. (2020: p. 36) note that in the context of firefighting “more dialogical forms... are less suited for... conditions of temporal uncertainty because they require time-consuming negotiation processes that are often not feasible.” Coordination efforts in these situations prioritize adaptability and immediacy of action to work collectively. Along these lines, research has sought to understand the coordination of work that is temporally uncertain (D. Geiger et al., 2020), contingent (Christianson, 2019; Klein et al., 2006), simultaneous (Du et al., 2020), extemporaneous (Ching et al., 2019; Pine & Mazmanian, 2017; Stephens, 2020), and necessitating fast response (Faraj & Xiao, 2006).

Coordinating in these conditions often requires member autonomy. When each episode of work, or work trajectory (Faraj & Xiao, 2006), is potentially unique, members tend to follow their own judgment. They may temporarily discard specialization-based boundaries, and enact flexible structures of joint sensemaking “matched to the information-processing demands of the environment” (Faraj & Xiao, 2006: 1156). Additionally, they may look for real-time cues indicating the contingencies of the present context. Research on improvised work in both

emergency (Faraj & Xiao, 2006; Geiger et al., 2020) and non-emergency settings (Ching et al., 2019; Stephens, 2020) finds actors coordinating through cues when they do not have adequate time to deliberate. In a study of ICU physicians, LeBaron et al. (2016) find that “deviations from expectations were useful, not detrimental” (p. 519), since, “when a breach of the participants’ shared expectations occurs, a variety of visible and audible behaviors was used to signal that something is problematic” (p. 528). Other studies find visual cues (Christianson, 2019), physical cues (D. Geiger et al., 2020), audible cues (Stephens, 2020), and tactile cues (Sergeeva et al., 2020) help in adapting to local contexts. Through sensing cues and improvising actions, members can work to coordinate effectively.

Difficulties with the Literature

The existing literature on coordination addresses the challenges of many settings, but it has not yet examined settings where algorithms— including data, code, and infrastructures— suffuse the work. In these settings, changes arise continually from algorithmic data that updates objectives, algorithmic code that updates directives, and algorithmic infrastructure that updates operations. Such settings are becoming increasingly widespread as cloud services become a primary means of delivering services. Yet the challenge of coordinating in such conditions of heterogenous, concurrent, and indefinite updates is not well understood. While studies of coordination have featured digital tools, such as email, shared repositories, and task management systems (Kellogg et al., 2006; Valentine et al., 2017), the functionality of these tools has been fairly stable and the focus of inquiry has generally been on how people use them. In contrast, live service algorithms are typically far from stable. When updating, they can potentially disrupt, interrupt, and change existing functionality in a moment’s notice.

I seek a more expansive view of coordination, one less centered on humans. In existing literature, digital technologies principally show up in roles assisting and mediating human interactions. These studies focus primarily on what people do with technologies. In their review, Okhuysen & Bechky (2009: 474) write that technological “objects and representations... offer a common referent around which *people* interact, align their work, and create shared meaning”. Accordingly, scholars highlight how, through discussions and planning sessions, people anticipate (W. C. Barley, 2015), articulate (Schakel et al., 2016), and deliberate (Bechky & Okhuysen, 2011) collective actions. And in studies of fast-response coordination, where people must adapt quickly, they show mostly human-centered, -initiated, and -driven adaptations (Christianson, 2019; Geiger et al., 2020; Stephens, 2020). For example, Geiger et al. (2020) writes about the rehearsals people go through to learn tacit cues. Others spotlight human activities involving reflective talk (Kremser & Blagoev, 2020), emergent dialogue (Faraj & Xiao, 2006), psychological states (Harrison & Rouse, 2014), emotional triggering (Stephens, 2020), and personal thanking, admonishing, and joking (Bechky, 2006). If research continues to center human activities, the unique contributions and priorities of algorithms, made possible by their memory, precision, and speed, may be overlooked in models of coordination.

Although research has acknowledged the potential of algorithms to exercise agency (R. S. Geiger, 2014; Shestakofsky, 2017; Valentine et al., 2017), the technologies actually studied in coordination research have been relatively fixed in function, representing a product at a snapshot of time after its release and disconnected from continuing development. For example, even though Kellogg et al. (2006) acknowledges that technology is “continually evolving, where new standards, protocols, coding languages, interfaces, security requirements, and applications were *released* frequently” (28), their study occurred between technology releases and did not capture

evolving updates. They examined calendar invitations, presentation documents, and status reports, all services offered through stable products which functioned in ways workers in their study intended and expected. The authors were able to conclude that coordination proceeds “without requiring all the specific details to be understood or worked out” (p. 31), because people control the changes and people can make sense of ambiguity. That conclusion does not apply with live cloud-serviced technologies, whose functionality are reprogrammable during use and continually updated through algorithms primarily, not people (Szyperski et al., 2002). In contrast to people, who can handle ambiguity, say by “heedful interrelating” (Stephens, 2020), algorithms operate “mindlessly” on precise instructions (Salovaara et al., 2019). “In simple terms, algorithms cannot reliably detect and act on events they have not been designed to handle” (Salovaara et al., 2019). If specific details are not understood or worked out, people might not understand what algorithms are doing, and algorithms might not respond as people intend (Christin, 2017b; Sachs, 2019). The closest example in the literature of people and algorithms updating work together is from Shestakofsky (2016)’s study, but his empirical setting was limited to “quick and dirty” decision-tree algorithms simple enough where “workers can stand in” (389). Going further, I want to capture the complexities of live service algorithms.

Existing accounts of coordination do not adequately consider the role and influence of algorithmic agency, but initial attempts in that direction show a path forward. Shestakofsky (2016) points out that “only by examining organizations in which software algorithms are being developed and implemented can researchers uncover the nature of human–software complementarities...” (383). Following this rationale, my study looks closely at the interactions between humans and algorithms as they develop and operate a live service through continual updates and, furthermore, how they coordinate those updates. My study is also informed by

insights by Actor Network Theorists who emphasize the substitutability of actions between human and technology (Callon, 1984; Latour, 1987). Most relatedly, Ribes et al. (2013), in their paper *Artifacts that organize*, turn scholars' attention to the potential of coordinating with algorithms, arguing that computational actors are capable of “delegating tasks and administering resources.” The term delegation is important, as it suggests “the in-principle interchangeability” between algorithms and humans “for the accomplishment of organizational ends” (Ribes et al., 2013: 4). These studies invite subsequent research to explain how people and algorithms might coordinate together. My study does this, studying a live service of global scale, wherein coordination efforts must deal with heterogenous, concurrent, and indefinite updates.

Research Methods

I collected data on the development and operations of a live service at an agricultural technology firm (DigitalAg). I focused on one specific service — a daily-updated imagery analytics service (zonal statistics) provided through one of the firm's digital platforms (AgMap). Below, I first describe the setting for my study — the firm, the platform, and the live service — before detailing my techniques of data collection and analysis.

Research Setting

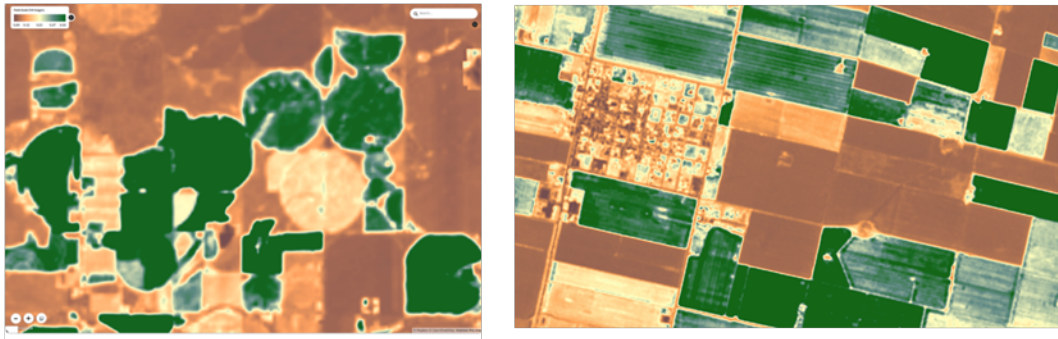
DigitalAg: The firm. For this research, I conducted over 15-months of non-participant observational fieldwork at DigitalAg, a fast-growing agriculture-technology company exceeding a thousand employees. Based out of a number of U.S. cities, DigitalAg runs agricultural operations in the Midwest, and its technology development primarily in the Northeast. Like other technology companies (i.e., Uber and AirBnB), DigitalAg seeks to transform traditional industries with digital services. Targeting agriculture specifically, DigitalAg develops algorithm-based services—including the data, code, and infrastructure that constitute them—to help

farmers and stakeholders (i.e., grain traders, food companies, and agronomists) trace, track, sell, transport, evaluate, and forecast crops. With ever growing scale, DigitalAg’s algorithms strive to be the system of record for global agriculture, food supply, and crop management. There being many offerings at DigitalAg, I focus on one of its platforms: AgMap, and concentrate on capturing the complexity of the work involved in sustaining and improving an algorithmically updated service.

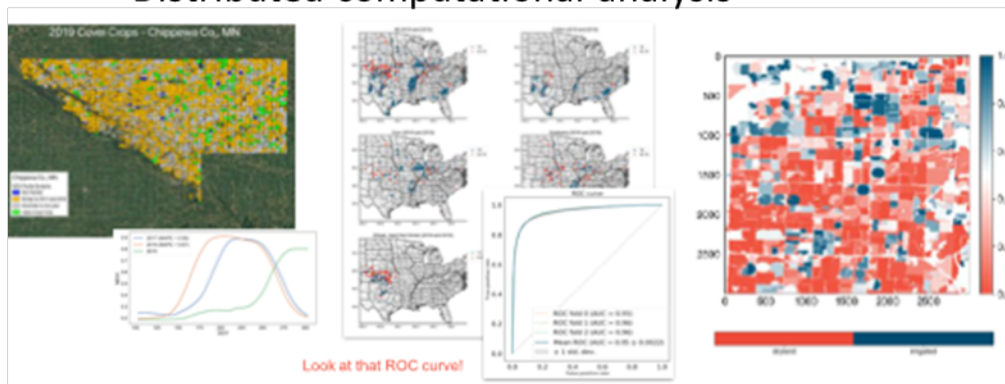
AgMap: The platform. Working out of DigitalAg’s Northeast office, roughly twenty-five members (and growing) are responsible for AgMap’s ongoing development. AgMap supplies real-time planetary-scale imagery and imagery-based services to DigitalAg’s other platforms as well as to external clients. AgMap’s core capability is remote sensing, an application of geospatial intelligence—using algorithms to analyze and visualize location-specific information commonly generated by drones, satellites, and ground sensors—that answers questions about activities on the surface of our planet (Nagaraj & Stern, 2020). They are part of a roughly eighty-member division specialized in geospatial intelligence. From its start, the purpose of AgMap has been to produce a ‘living’ map of global food production (their term), offering real-time imagery, insights, analyses, and forecasts on crop conditions. With imagery covering more than five billion acres of farmland going back twenty years, and real-time forecasts of future crop activities, AgMap helps users monitor crop-health and yields, map and route supply chains, track farm property values and crop prices. AgMap’s services include monthly reports, regular briefings, webinars, daily-updated maps and dashboards, and platform data accessible by API (application program interface) and web interfaces (see **Figure 1** for visual examples of AgMap’s continually updated service). Outputs constantly change with new data, code, and

infrastructure updates. Despite that, AgMap aims for the consistent and continuous production of accurate, real-time, planetary-scale imagery and analytics.

Petabytes of satellite imagery



Distributed computational analysis



Portfolio of models & reports

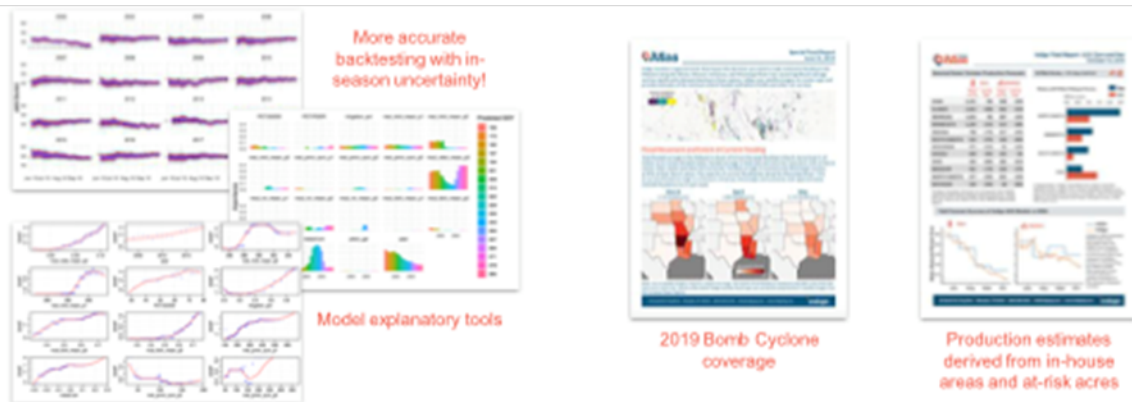


Figure 1: Outputs of the AgMap platform

Like many algorithm-based services, AgMap is built of modular components (Bartholet et al., 2004). In fact, members refer to the zonal statistic algorithms as a “pixel factory”¹ comprised of live data, code, and infrastructure components (See **Table 2** for examples of AgMap’s algorithm components). Most components could be developed independently of others, and have “off-the-shelf” functionality that could be integrated relatively easily into other code (Szyperski, 2003) and updated separately. Each component follows its own ‘terms’ of service, and the resulting mix of updates aggregates up to update the zonal statistics service. Members focus on improving the overall “factory,” and count on the components to remain functional.

Algorithm components	A few examples of externally provided components	Description	Acronyms
Data	NASA’s Landsat Imagery NASA’s MODIS Imagery ESA’s Sentinel Imagery GFS Weather U of I Meteorology ECMWF USDA’s NASS	Provides various satellite imagery data streams (i.e. crop cover) Provides various weather data streams (i.e. cloud cover, precipitation) Provides various crop data streams (i.e. area planted, yields)	NASA: National Aeronautics and Space Administration ESA: European Space Agency GFS: Global Forecast System U of I: University of Iowa ECMWF: European centre for Medium-Range Weather Forecasts USDA: Department of Agriculture NASS: National Agricultural Statistics Service
Code	Mapbox GL GDAL (rasterIO, xarray) Numpy GEOS (shapely, Fiona) Tippecanoe	Provides mapping functionality Provides image processing functionality Provides statistical functionality Provides image display functionality Provides image display functionality	GL (OpenGL): Graphics Library GDAL: Geospatial Data Abstraction Library GEOS: Geometry Engine - Open Source
Infrastructure	SALT AWS EC2 AWS S3 GCP Snowflake GIT	Provides cloud management control Provides cloud processing capability Provides cloud storage Provides cloud storage Provides data warehousing service Provides code management control	AWS: Amazon Web Services EC2: Elastic Compute 2 S3: Simple Storage Service GCP: Google Cloud Platform

Table 2. Examples of algorithm components which continually update in live service

¹ Pixel factory, so called because its machinery processes pixel-based imagery.

Heterogenous, concurrent, and indefinite updates make this work difficult. First, AgMap’s updates are heterogeneous, and follow a mix of directives. For instance, AgMap’s imagery builds on public-access data sources (e.g., NASA, USDA, GFS), open-source code (e.g., Pandas, Rasterio), and third-party infrastructure (e.g., AWS, GCP) and integrations (e.g., Git, Docker). Even within AgMap, members work with each other’s modular components, and may have little awareness of updates inside those components. Second, updates are concurrent, meaning that updates seldom need to wait for acknowledgment or change approval. They can modify algorithms already in use. Third, service updates may arise indefinitely, with repairs and open-ended improvements continuing throughout the life of service.

To make matters more complicated, updates occur over what members call ‘a planetary scale’ of production. “How do you make sense of four billion data points?” an engineer once asked me jokingly. The size, speed, and variety of operations can be overwhelming. On one occasion, an engineer said:

If you add up the tasks for each tile [a unit of imagery], for each satellite product [the source of imagery], for each date, for each batch of files, that’s 2.4 million tasks [computational tasks] total just for this [one] job [operation].

AgMap provides many services, but there is one in particular that is core to the platform: the development and delivery of zonal statistics. Explaining multiple services would risk confusion without yielding additional insight on my research question, so this study centers on one.

Zonal statistics: The service. The daily production of *zonal statistics* is the core work that converts imagery into usable form for downstream analytics, forecasts, and visuals (see **Figure 2** for examples of downstream outputs). As such, zonal statistics feed the rest of AgMap. Before explaining the zonal statistic, it is important to understand what imagery is, specifically, digitized imagery. Digitized imagery, also known as rasters, comprise pixels of information that

each represent snippets of the world. Rasters cover a geographic area, at a specific time, and depict a metric of interest. Important metrics for visualizing include atmospheric weather, ground surface crop health, and cloud cover. Each of these may derive from a different data source, reflecting different parameters and dates. Multiple rasters are layered together to form meaningful visuals. By layering rasters, and processing that composite imagery, AgMap can then extract data out of the imagery for downstream analyses, predictive models, and visual displays. That data are zonal statistics. Although DigitalAg has a proprietary method of producing zonal statistics, at its core is a basic GIS (geographic information systems) operation used widely by companies and academics worldwide (Bolton & Friedl, 2013).

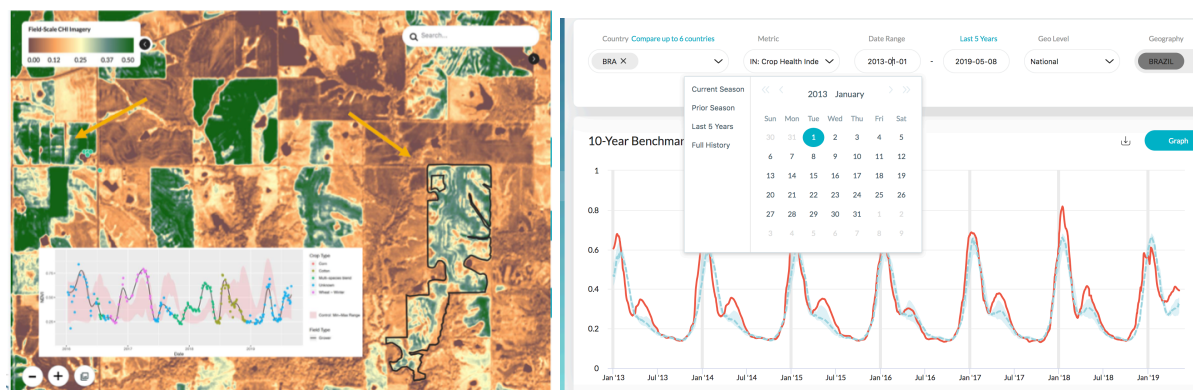


Figure 2: Example of AgMap analytics, forecasts, and visuals

Zonal statistics offer a way to summarize data across the multiple images (i.e., cloud, weather, and cropland images of the same zone). AgMap summarizes the relevant images covering a particular geographic area into a set of statistics, and uses those in its analytics, models, and maps. Zonal statistics does the work of translating image pixels into numerical metrics. Hence, zonal statistics are statistical summaries of images of specific geographic zones for a given metric. The procedure for making zonal statistics varies, depending on the targeted zone (e.g., which regions to include), the chosen statistic (e.g., which kind of mean), the desired

metric (e.g., which kind of crop health index or weather index), and the source of data and code used, but all involve the same general process. Specifically, it follows this sequence: ingesting imagery from many data sources, finding imagery files for the areas of interest, matching and stitching them together, applying necessary transformations (e.g., projections), filtering out unwanted signals (e.g., clouds, non-crop vegetation), and summarizing the image in statistics. People and algorithms do this work together through the cloud to produce daily zonal statistics.

It is worth emphasizing that zonal statistics are frequently and extensively updated in order to provide users with fresh analytics using the latest algorithms. Constituting these algorithms are numerous databases (i.e., imagery data), codebases (i.e., image processing code) and computational infrastructure (i.e., cloud servers), all entangled in practice. Many people and algorithms, inside and outside DigitalAg, continually update these components, replacing data, rewriting code, and reconfiguring servers. At any given moment, the zonal statistic can change. For example, one day, the code calculated a mean statistic using a new definition of the bounding area, smaller than previously used, and the pixel gradient of the resulting image changed, surprising users with coloration different than all the previous days' imagery.

In fact, any particular algorithm component, whether data or code, can act unexpectedly even if it does not itself change, entangled as it is with other data and code that may change. Satellite imagery, for example, being too big to store in a single data file on one's computer, is actually generated piece-meal: quality-checked, and assembled through code at the moment of use. Data outputs can change when the code rendering them changes, without any change to the input database. Code also takes actions not explicitly written in codebases, since they are shaped, parameterized, orchestrated, and trained by ever-changing data. The actions of code can be valid in one moment and invalid the next when the benchmark data updates. Hence, zonal statistic

production is not an assembly of things (just data or just code), but an entangled algorithm (of data and code executing on specific infrastructure) whose actions continually update (this illustrates Kwan (2016)'s notion of algorithmic uncertainty). A director of operations expressed:

The company believes there's all this data, and we just gotta go get it. But there's nothing there to get. We have to make it. So much work goes into getting the data workable [and we have to do it every day].

The complexity constituting the zonal statistic service is similar to other algorithm-based services that also exist through heterogenous, concurrent, and indefinite updates.

Data Collection

My observations of AgMap, starting October 2018, followed a longitudinal research design (Fine et al., 2009). I had a desk at DigitalAg's northeast office, and visited roughly three days a week, four hours a day, for more than 15 months. In addition to full building access, I was granted SSO (single sign on) access to the company email, internal wikis, work management trackers, product platforms, and Slack channels. To become familiar with the agricultural side of the business, I spent over 50 hours during the summer of 2019 visiting farms and observing the Midwest office. Venturing beyond AgMap, I attended meetings of other teams and projects at DigitalAg to better grasp the context of the work. Despite the unique technical aspects of AgMap's imagery-based algorithms, I saw that the coordination challenge of developing a live service was similarly felt by other platforms. By January 2020, I shifted attention from data gathering to analysis and writing. For another year, until Dec 2020, I studied AgMap virtually, through digital traces and weekly all-hands meetings via Zoom. Aware of the difficulties of studying algorithm-based work ethnographically (Christin, 2020; Seaver, 2017), I drew on a large variety of data sources to help see, track, and understand what was happening (see **Table 3** for descriptions of data collected). I briefly describe these data sources.

Semi-structured interviews were an important source of evidence (Weiss, 1995), the majority of which were conducted in person, one-on-one, and more than once. Because AgMap’s work was highly complex, largely invisible, and frequently changing, I interviewed people multiple times a quarter, with their computers beside them, to follow changing conditions and understand their thinking. Initial interviews asked about the overall work, and later interviews probed for responses, interpretations, and intentions at particular moments (i.e., before making an update, or after an update disruption). I conducted more than 100 scheduled interviews, covering every member of AgMap (as of December 2019) at least twice, each interview lasting between 45 minutes and 2 hours, all of which were audio recorded and transcribed (see **Table 3** for details about informants and interview questions). As all interviewees expressed interest in the research, they shared enthusiastically, generously, and patiently despite their busy schedules.

On-site and virtual observations were also crucial (Spradley, 2016). I logged over 500 hours of in-office observation time, mostly spread between meetings and work shadowing (where I pulled up a chair and sat next to members at their desks for several hours a day as they worked). Because the actual production of zonal statistics is, like other digital work, largely “invisible” (Cetina & Bruegger, 2002; Star & Strauss, 1999), carried out behind screens and in remote servers, I took advantage of DigitalAg’s abundance of meetings to hear people discuss and describe what they did, what had happened, and what they planned to do. In most meetings, workers displayed their screens, gave demos, drew on whiteboards, or walked through their workflow step by step. I attended over 50 weekly DigitalAg all-hands meetings and 11 monthly AgMap all-hands meetings, as well as hundreds of other, smaller meetings: daily stand-up meetings, code review meetings, sprint planning meetings, and impromptu team meetings. These were supplemented with casual conversations (Becker, 2008) during in-office lunches and coffee

breaks, as well as outside of the office, over board games, runs, happy hours, and parties. I recorded field notes, pictures, and diagrams on a laptop while at the office, and wrote up memos and summaries afterwards.

Data Collection		
Type	Description	Use in Analysis
Semistructured interviews <ul style="list-style-type: none"> Members: 38 Interviews: 102 Duration: 35 – 95 minutes 	Entire AgMap team as of 2020 was interviewed, including front-end developers, software engineers, data engineers, data scientists, data quality managers, geospatial analysts, remote sensing scientists, operations manager, and product directors, plus affiliated developers and users	Explore the tasks and goals of the members and the kinds of work they did. Identify the interpretations, feelings, and expectations members had of their work and those whose work impacted them.
In-person observations <ul style="list-style-type: none"> Total: 500 hours (approximately) 	Meetings (100+): I attended daily standups, weekly code reviews, bi-weekly sprint planning, weekly all-hands, demos, post-mortems, cross-functions, product emergency meetings Work shadow (60+ days): I sat beside every member of AgMap, at their desk, as they worked, for at least three hours each, and also had my own desk in the office	Examine the timing and patterns of activities, whether planned or impromptu. Understand the mechanisms, in the work, that raised urgency, drew collective attention, and drew independent attention.
Virtual observations <ul style="list-style-type: none"> Time frame: 15 months 	Algorithm management tools: code commit tracking dashboard, cloud computing dashboard, task management dashboard, platform activity tracker Messaging channels: 1,000+ bot (alert) messages, 75,000+ human messages	Analyze the interactions members had with algorithms to discern themes in their pattern of activity Understand the factors in their work that made coordination difficult.
Archival materials Total: 346 documents	Within AgMap: wikis, digital lab notebooks, code ticket histories, code documentations, product descriptions, presentations, post-mortems, data request logs, production schemas, screenshots, mockups, performance reviews, company documents Outside AgMap: open source code documents, public data document, software vendor documents, community forums, technical guides, industry blogs	Code the descriptions and explanations given by members and others about how and why algorithms were updating Categorize the experiences and rationales members faced when updates were made by them or by others.

Table 3. Data collection sources and descriptions

Through interviews and observations, I obtained insight into people’s actions and intentions, but, as scholars have acknowledged, it is seldom feasible for technical and security

reasons to examine people's interactions with algorithms (Christin, 2017a; Pachidi et al., 2020; Sachs, 2019). In this regard, my access at AgMap was uncommon and valuable. While restricted from the full codebase and data entries, I got a pulse on real-time algorithm activity by checking digital traces to confirm what I heard and observed, and noticing what was being done and when. Examples of digital traces included time-stamped actions logged in digital lab notebooks, ticket histories tracked on a task management platform, documentations in blogs and internal wikis, records kept by third-party services, and presentation slides and Word documents. Additionally, I monitored 28 Slack channels, reading over 75,000 messages sent between people, as well as the thousands of bot-alerts in automated Slack channels.

Archival materials helped me make sense of technical details. I consulted technical guides, online community forums, internal algorithm design schematics, code re-pastes, and GIT comment histories.² My transcripts, notes, memos, and documents were organized in Atlas.ti. I describe next how these were analyzed.

Data Analysis

I adopted an inductive qualitative approach, iterating between data collection and analysis (Langley et al. 2013), to understand how development of a live service is coordinated in conditions of heterogenous, concurrent, and indefinite updating. Analysis unfolded using multiple data sources: repeated interviews with every member of the team; my own in-person and online observations of the algorithmically-mediated work; and archival materials documenting and explaining aspects of the service (Glaser and Strauss, 1999; Charmaz, 2006).

My analysis required a different analytical approach than scholars typically use to investigate technology-based change. Research has tended to organize analyses around a launch

² Due to sensitivity of this information, I do not display them directly but they informed my analyses.

event, like the introduction of new machinery (S. R. Barley, 1986), or the release of new features (Shestakofsky, 2017). But the live service in this study changed frequently, its technology updating constantly, well after the service was launched. Scholars examining micro-level changes in a service (Langley et al., 2013) have advocated for a “trajectory of care” approach (Faraj & Xiao, 2006; Christianson, 2019), capable of capturing the unfolding of a service and “the total organization of work done over the course” of the service (Christianson, 2019: 52), which allows for comparison of outcomes across service trajectories (e.g., treating and discharging different patients). The live service in this study, however, went on indefinitely, the same components of service being developed two years after its release to users. Unable to compare trajectories, I examined a single ongoing service for over a year, and compared moments when it was disrupted and not. My unit of analysis was work practices, the recurrent, material, and situated activities (Feldman & Orlikowski, 2011) that AgMap members and algorithms enacted to keep the service continuous and consistent.

I began the research with a focus on how members assessed the “quality” of the algorithmic service in development, but after an initial coding of my field notes and interview transcripts (Charmaz 2006), a new focus emerged. When I open coded the interview data and archival data, two things stood out. One, that members were continually concerned about use of the live service, and two, that members continually assessed aspects of the already-launched service as unacceptable. This initial finding surprised me, since it indicated that development could be good enough for users at one moment yet unacceptable at a later moment. To validate the findings in my observations of the work, I open coded data collected in my field notes and in online conversations among members. These data showed real-time activities and rationales, and were important in corroborating several themes.

Specifically, the open coding of observational data showed that members were frequently concerned about keeping the service live. Surprisingly, their concerns did not diminish as development went on. To understand what keeping service live entailed, I went through my interview transcripts, company documents, and any notes of conversations regarding ‘live,’ ‘living,’ or ‘current’ service. I found that the dominant criteria for a live service to be considered acceptable was that it operates continuously and consistently, where its current operations are assessed against its own prior operations (i.e., crop health metrics today compared with yesterday). I looked for corresponding evidence in archival materials, and found self-referencing indicators of acceptability appearing in AgMap’s review meetings, Slack conversations, post-mortems, and code criteria. This step was crucial because it highlighted a key insight—the cause of disruptions—revealing that a live service could turn from acceptable to unacceptable due to updates. To assess whether this analysis triangulated with other data sources, I systematically went through documents, presentations, and interview transcripts that described the composition and architecture of the live service. I found members discussing the use of numerous components updating concurrently through the cloud, and attributing this updating behavior as the reason a functioning service might lose continuity and consistency.

To identify the specific activities members engaged in to keep a service live through update disruptions, I went back to my field notes of in-person interactions, records of virtual interactions, and archival materials about the algorithmic service, to do more focused coding (Charmaz, 2006). I looked for moments of disruptions arising from updating algorithmic components, and coded for instances before, during, and after disruptions (e.g., disruptions could arise when code produced error, when data was discrepant, or when a cloud computer crashed), where members either talked about mitigating, reducing, resolving, or preventing disruptions, or

showed signs of doing so in traces of algorithmic activity. When I saw indication of members anticipating a disruption in a forthcoming update, and proceeding with the update, I coded those activities as incorporating improvements. At times, disruptions arose unforeseen, especially when updates originated beyond AgMap. I observed activities intended to detect whether those unexpected updates generated disruptions, and I coded the activities as detecting disruptions. I also observed activities intending to re-establish the continuity and consistency of service in the wake of a disruption, which I coded as restoring functionality. In further coding, I observed that not only members, but algorithms too, contributed crucially to these practices. This step led me to identify sub-practices leading to each set of activities.

To refine my analysis, I re-examined all the data to identify how members experienced disruptions, looking at the ways they described moments of disruption. Doing this helped me identify variation in members' reactions which I had not found in their activity traces. When coding their reactions, I noticed seemingly contradictory themes, depending largely on whether the reactions were expressed during a disruption, or afterwards. Members reported in meeting notes and online chats of being frustrated, stressed, and annoyed with disruptions, and I found these sentiments concentrated at two moments during disruptions: soon after disruptions were detected, and prior to disruptions being resolved. Because this transcript data suggested a collective desire to prevent or minimize future disruptions, I was confused by the activity data revealing the continual emergence of disruptions, seemingly increasing in frequency. As I tried to understand why disruptions kept arising despite members' explicit frustrations when dealing with them, I looked more broadly at my data, specifically at how members dealt with updates prior to disruptions arising. This step helped me see the importance of disruptions for accelerating updates. Because my archival data revealed that members were congratulated,

rewarded, and pushed to make updates quickly, I coded instances where AgMap introduced updates for which the risk of disruption was explicitly articulated yet not fully addressed. Because they pushed update after update, and found in these updates known risk of disruption, I coded these activities as perpetuating disruption, and found it to be widespread as I triangulated across my data sources.

To develop a theoretical account of my findings, and explain their relevance to other settings of work, I considered what perpetuating disruptions accomplished for AgMap. As I did this, I iterated between the findings and existing literature, to make sense of how perpetuating disruptions could, counterintuitively, be a way to achieve continuity and consistency of a live service that is collectively updated. This process suggested that my preliminary conclusions connected to the coordination literature. I noticed that relative to my findings, past studies largely described coordination as a deliberative process leading to certain or provisional ends. These studies did not adequately account for what I had observed. Synthesizing insights from the literature with my findings, I noticed that the coordination process I observed was novel in that its aim was to produce an incomplete yet always improving service. For example, engineers used language like “it’s never done” and “we are chasing completeness” to describe their collective work. When I revisited my findings, analyzing my data with concepts of incompleteness in mind, it revealed the role of workers’ practices in coordinating a self-reinforcing process by which updates perpetuated disruptions, and disruptions accelerated updates. As my data highlighted the importance of keeping the service live, I theorized these coordination practices as “regenerative” practices. The model that I explicate in the following sections helped me see how the work required to keep a live service going entails coordinating perpetual updates and disruptions, a

form of coordination that contrasts with that in the existing literature which highlights processes proceeding deliberately towards settled ends.

Findings

I first highlight the overarching priority influencing workers when developing a live service: to keep the service operating continually and consistently while improving content through in-service updates. Following this, I elaborate on three coordination practices enacted by workers to keep operations improving despite unavoidable disruptions emerging from updates. Specifically, I articulate how, by *incorporating improvements*, *detecting disruptions*, and *restoring functionality*, workers move live operations into disruptive states and then quickly out of them. I describe the advantages and drawbacks of coordinating in disruptive states, where the live service can thrive by staying live through urgent disruptions. Furthermore, I show that workers, bolstered by these coordination practices, may perpetuate disruptions that leverage moments of disruption to grow the service, and I detail the consequences of this.

Priority of Keeping Operations Continual and Consistent

Users of a live service expect content to continually update, its issues to be fixed and operations to be improved not once, but repeatedly, throughout the life of the service. I observed AgMap promising as much, in client-facing reports, in internal presentations and documents, and in conversations among AgMap workers. In one of AgMap's product presentations, describing the outlook for the service, a list of continual improvements in imagery data, processing code, and computing infrastructure included: "greater frequency, lower latency, finer resolution, fewer clouds, better quality, more sources, [and] global coverage." AgMap's commitment to improving content was evident in my field notes and archival materials. Nearly every week, either in a daily standup meeting or Slack conversations, members identified additional data sources, proposed

new code configurations, or described alternative cloud implementations intended to improve operations. I tracked this update activity in members' code commits and task management dashboard. Improvements were demoed in daily stand-up meetings and monthly all-hands meetings, and also showed up on the monthly reports to clients, weekly webinars, and the AgMap platform's daily dashboard.

With each improvement, came the risk of unintentionally disrupting live operations. Concerned about user complaints, AgMap prioritized dealing with disruptions. Everyone at AgMap acknowledged that continuity of operations was crucial. "We can't afford to have the app [the service] not work for a day," a member reiterated. Their overarching goal was to ensure operations remain accessible for use and functional at every moment, no matter who was using the service, for what purpose, or the form of output. Indeed, a key part of usability rested on the consistency of outputs. My field notes indicated that in the majority of cases where the service failed, or was deemed problematic, it was not that operations completely failed to function, but that operations ran with altered functionality, resulting in outputs that deviated questionably from previous ones. As a data scientist put it: "[Users] will have a prior, you don't want to compete with too much," explaining that an update could be technically valid and yet be disruptive if users could no longer use the live service consistently. It had initially seemed, from my early interviews, that achieving continuity and consistency would be straightforward, since, as the head of engineering said: "We make imagery. We are doing the same thing again and again, so it repeats itself."

Within a few weeks of my study, I began noticing records of service disruptions, commonplace in archival materials and digital logs, that told a more complicated story. A data scientist recounted one such "traumatic" moment:

We had something [in production] that kept running, and it should not have. Yea, it was on a schedule... running every day... and it never updated to new data, and it just kept infilling into the future.

The disrupted operations continued to yield outputs appearing within “acceptable bounds,” and so was not noticeably problematic. He continued, voice rising: “and then, you realize, Oh! this has been broken for three weeks before we realized that other things are going on!”

It turns out that data were being updated, but no corresponding update was made in the code that used them, thus producing the disruptions. These kinds of disruptions were unavoidable since modular components were intentionally updated independent of one another. AgMap workers were not alone in updating components of data, code, and infrastructure. Other contributors, including other parts of DigitalAg, and external organizations such as data providers, open code communities, and cloud-service vendors, as well as the satellites downlinking daily imagery, all provided concurrent updates. This explains how a service could repeat itself, leading users to expect continuous and consistent service, and yet be disrupted at any time by fresh data, modified code, or new infrastructure settings.

To summarize, keeping operations consistent and continual while improving them is challenging because updates intended to improve a component for a particular purpose at a particular moment, can on other occasions turn disruptive. Even though workers were cautious incorporating improvements, some disruptions were unavoidable and required workers to respond quickly. But which updates would turn disruptive, or when, was impossible to fully predict or prevent, given that cloud-based components were open to concurrent and indefinite updates from multiple sources. Even to learn of updates and find disruptions in modular components was a challenge, since update details were seldom announced in advance. Moreover, in the time it took to recover operations, users still expected the live service be usable. In light of

these challenging conditions, an important question concerns how workers can coordinate to keep the live service going. In the following, I present the three coordination practices that I found AgMap workers enacting to address this challenge: incorporating improvements, detecting disruptions, and restoring functionality.

Incorporating Improvements

AgMap had been providing usable live service since its launch, and yet, improvements of varying kinds were incorporated practically every day. More resilient and robust code, more accurate and fine-grained data, and more efficient and explicable operations were among the improvements released into live operations through continual updating. “It’s never ever finished, [there’s] always a next one [another update],” the head of engineering observed. AgMap’s operations did not stop for updates, and carried on producing real-time imagery and analytics, refreshing platform dashboards and output feeds. In short, workers continued to modify code, refresh data, and reconfigure infrastructural components of the service while providing the service live to users. To mitigate the risk of disruptions, workers incorporated improvements through two sub-practices that I call *initiating improvements* and *introducing improvements*.

Initiating improvements was an individually-directed³ set of activities for achieving algorithmic updates. AgMap was constituted by modular algorithmic components, meaning that individual components of an algorithmic operation could be updated independently of other components. For example, workers could update particular data without accessing or editing any other data or any of the code that processes them. Because components interacted through their outputs, workers could improvise updates to some components without much concern for what

³ Individual here refers to the specific component updated, which could involve more than a single individual person. Modular components tend to be attended to by only a small subset of those affected by them, which could be one individual or a small group.

was being done to other components, so long as the outputs of all components remained usable. Though updates affected all users, updates were pushed forward primarily through individual initiative and, only later, during or after implementation, were updates announced or discussed in detail. This separability allowed for a “clean divisions of labor” as an engineer noted, and it accelerated service updates.

In talking with members, I learned that the responsibility for initiating updates to specific components—for searching out, testing, and detailing improvements—was assigned to designated members. On AgMap’s organization chart, each member was considered the “lead” or “expert” on particular components of the service (i.e., crop data lead, meteorology data lead, flooding data lead), and responsible for staying on top of the updates and update opportunities of those components. For instance, members followed different vendor blogs and news alerts, read different FAQs and forums about their data and code dependencies, and kept tabs on the live activity of their components, searching out new data, code, and implementation updates for AgMap. Stand-up meetings were arranged each morning where each member could briefly mention what they were independently updating. Unless there was a disruption to discuss, this typically took no more than a minute per person.

Management encouraged members to direct their own improvements and kept goals for the team deliberately open-ended. For instance, the goals listed on an engineer’s quarterly review document included “improve [user] confidence in data quality and completeness” and “develop more accurate outputs.” Leaving the specifics open allowed members to initiate the improvements they considered most important, at the time they felt most appropriate. Doing so was more than encouraged, it was an explicit norm. AgMap’s official list of core expectations for members indicated:

[The] expectation is that you make your own decisions quickly and you take ownership of those decisions. Do not be afraid to make a call. You were hired to make shit happen.

Accordingly, members exercised personal discretion in modifying data, editing code, and adjusting infrastructural settings, without having to first align the details with others. For example, individuals frequently turned off or “snoozed” certain code components they considered to be unnecessary or “not a big deal.” They also transformed data components using new filters or parameters deemed preferable at a given moment. After each update, they explained to the rest of the team what they had done and what vulnerabilities they may have introduced.

Initiating improvements proactively was seen as desirable, but so was updating cautiously. Largely unaware of how others were using the live components, workers made updates that could inadvertently interfere and disrupt existing operations. To mitigate that risk, members and algorithms separated the improvements they initiated and the live operations they sought to update, safeguarding users temporarily until the improvement could be tested. Many of AgMap’s techniques for separating individual initiatives were standard in software development practice. One approach was to work in “staging environments”—workspaces cut off from live operations, in which code and data can be manipulated and tested as if isolated “off-stage.” An analogy would be to working on a Google doc off-line. Members referred to staging environments as “protected,” “safe,” and “without risk.” An engineer explained, “This is to help make sure that [our] work... does not interfere with the [outputs] being used by [users] in production.”

Another approach was to take components from the shared “cloud,” create copies (“mirrors”) of them, and run these on individual computers where changes could be locally

confined. A third approach was to work within “sandboxes.” These are personalized versions of an otherwise shared program in which members could use their preferred programming languages and configurations, as if playing in their “own personal playground,” until they were ready to introduce the improvements into the live system. This separability accelerated updates. The head of engineering said, “Everyone has their own processes for doing [things]. It allows [them] to move fast.” Expressing a similar sentiment, a data scientist said: “it is much easier to be a lone ranger... easier to put my head down and not share ideas and expectations of how things should be done.”

I found in interviews and archival materials that members worked with data in personal styles (i.e., short-hands and hotkeys), wrote code in their preferred programming languages (i.e., Python or Java), and curated individualized computing environments (i.e., on desktop or virtual computer). Though such separability eased individual efforts, the resulting changes were not always compatible with live operations, and would later require additional work to translate (i.e., rewriting code from an individual’s preferred language into the language used by AgMap’s shared systems).

Developing updates in complete isolation is problematic since a component tested without live inputs might not function properly when brought into live service. That is, improvements of one component may pass tests in insulated environments but turn out to be incompatible with the updated outputs of other components. An example is code that may have been independently improved yet not adjusted for live data. Consider what happened in the case of a piece of code that converted satellite imagery pixels to crop health metrics using an improved pixel zoning technique. A sandbox version of the code yielded improved results when processing historic imagery, but then produced anomalous results when fed new imagery

containing previously unobserved cloud patterns and color distributions. Another example, with data, occurred when crop mask data was cleaned and reformatted but no longer fit with the parameters and logics of the code that processed it, which had been updated separately.

Rather than isolate components, members tried when they could to initiate improvements within surrogate (as opposed to live) operations. Surrogate operations were snapshots of live operations but did not affect live users. Surrogate operations comprised all the relevant dependencies and configurations affecting a component, but in “synthetic” form (as members put it). Synthetic components included mirrored computational resources, data copies, proxy code, simulated computing environments, most-recent versions, and other mocked-up versions of dependencies. Initiating improvements within functionally separate surrogate operations made the work faster and less expensive to do. A member explained,

Data for the secondary environment [staging environment] should simply be copied from [previous operation], not provided via [live operations]; this reduces costs and complexity of the solution.

Introducing improvements was a similarly individually-directed set of activities which brought separately initiated improvements into live operation. Once workers were satisfied with component updates, having tested them in insulated environments, they took steps to introduce them live, updating the service for all users. Members referred to this transition—from the testing environment to the live environment—as “pushing to production,” “taking [production] out of staging,” and “publishing to users.” Technically, it was accomplished in several ways, but basically it released the improved component into live operation, akin to pressing “save” on a Word doc. More specifically, it took the ‘sandbox’ version, previously accessible to the individuals updating it, and synchronized it with the live version operating through the cloud, where it reached all the service’s users. To continue the analogy, this is akin to “saving” the

Word document to a shared DropBox folder. Relevant documentation usually accompanied these updates, detailing the improvements that had been made.

Most improvements refined existing components incrementally, and were intended to enhance, not fundamentally transform, current operations, so as to ensure consistency with what came before. For example, particular kinds of data in use might remain (i.e., temperature data), but the data source could change to offer better precision or higher resolution. Similarly, with code, changes were made to improve processing speeds of existing operations, not to perform unrecognizably different operations. As long as components functioned consistently, they could be independently and concurrently updated. Members told me that by counting on consistency, they could ignore the details of any particular update if the improved component was still functioning as expected.

Apart from a passing mention in a meeting, or a nominal post on shared channels, I hardly ever observed members announcing relevant details of their improvements. Rather, I found that members documented their work and referred colleagues to these materials (e.g., their methods, reasons, and limitations for specific changes), which were kept up-to-date in wikis, task tickets, digital notebook entries, and automated logs that could be accessed and read at any time. In one case, two members identified an issue, began implementing updates, and informed others about them through documentation only after the improvement had been introduced. One of them wrote: “We found the issue. [The other engineer] is [changing] everything now... We'll follow up in the [documentation].” By using documentations in lieu of direct correspondence and meetings, they could introduce improvements faster, as two engineers noted:

[In] working in a fast-paced environment... when you need to communicate to [other] people, you better have a good wikipage and visuals so people can understand... all the documentation is there so that the machine learning engineer doesn't need me all the time... everything's online and documented.

We keep doing different things... and [the documentation] tells us what's happening... what we're doing now. I don't even need to tell [Coworker]. He'll see it [in the documentation].

Because improvements were introduced in a relatively independent and inconspicuous way, there was a high potential for disruptions despite the cautions workers took. While each component functioned well in testing, its outputs appearing consistent *prima facie*, it could still disrupt live operations if its “improved” function no longer fit with the live system. Since the live system changed constantly, even the most accurate surrogate operation at a given moment was insufficient. As a simple example, consider the following change made one day to a data component. AgMap was designed to expect a new batch of imagery daily, arriving in the same file location always by 10 AM. This happened every day for months, until one day, it unexpectedly arrived at 2 PM instead. The update to data delivery time was immediate, initiated by the imagery provider through the “cloud” pipeline. News of the update however came much later. In the meantime, the service continued to operate, disrupting downstream services, such as live data models whose use of zonal statistics assumed same-day imagery but unknowingly got day-lagged imagery instead. Workers could not have accounted for this change in the system when introducing their updates. One might argue for a less input-sensitive procedure (i.e., do not specify a time cut-off), but it was in fact sensible, practical, and normal for members to hard-code certain parameters, as modular components were expected to remain consistent in function. As a member pointed out, many changes were not foreseeable until they occurred: “We tend to take shortcuts as far as hardcoding things.... in the future you're going to have to clean that up, but [you don't know when].”

One could also argue that this update ‘blindness’ is only relevant for components updated by external providers, but I found that many of the improvements that disrupted AgMap

operations were internally introduced. Colleagues did not always think it necessary to communicate a slight change, or were unable to notify others right away. While most improvements were not in themselves conspicuous, disruptions to live operations were likely to be noticed. However, disruptions often took longer than desirable to notice, since most users could not anticipate them or learn about the associated update until after it was documented.

In summary, members and algorithms initiated and introduced improvements with considerable independence and autonomy, supported by organizational norms and technical arrangements that accelerated and safeguarded such pursuits. This enabled members to pursue specific goals of their own making in lieu of the open-ended goals assigned by management or vague requests by users, and to work quickly, in a manner they felt most confident, but this also exposed live operations to unforeseeable disruptions.

Detecting Disruptions

At least once every week, I witnessed a disruption in live service, observed members express alarm about a disruption in standup meetings, or saw alerts about a disruption on Slack. Typically, disruptions in live operation were triggered by updates intended to improve a component, whose details most users had been unaware of until the update was introduced. Components that had been operating without issue were effectively “black-boxed” during operation, so if something changed, the change would not be widely known. It was not that inner details were inaccessible,⁴ but rather that neither the requisite time nor attention was usually justified to keep checking for changes. Without a definite timeframe in which to expect updates, or a specified set of targets to look out for, there was too much to inspect by eye. Nonetheless, update awareness was crucial, in case disruptions arose. Reflecting on one such incident, in the

⁴ Many data and code components were open to viewing or at least well-documented

wake of a particularly disruptive update, a member said: “The [operation] has not run for 3 days. But we don’t have an error [error code] for that. We have to get errors [error codes].”

By engaging in a set of activities that I label *detecting disruptions*, members and algorithms created conditions capable of notifying when updates turned disruptive. First, they *defined disruptions*, instructing algorithms to differentiate normal from disrupted operations. Second, they *displayed disruptions*, prompting algorithms to catch disruptions and alert those able to respond. By detecting disruptions through these two sub-practices, members kept a pulse on updates without having to inspect every component all the time in a search for issues.

Defining disruptions clearly and reproducibly was crucial for aligning algorithms to members’ expectations. And including algorithms in the work was necessary, since people alone could not keep up with the scale and pace of operations. There were too many updating components, compounded by the indefinite period for updating. To include algorithms however, activities needed precise definitions, for, unlike people who might be able to sense if something (i.e., numbers or images) looked wrong, algorithms were ‘mindless,’ as a member emphasized:

[Algorithms] can only alert you on stuff that you actually made metrics for. So, if you don't instrument your stuff to give the right metrics, it won't do anything.

We found members furnishing algorithms with specific criteria for discerning disrupted activities. Based on the algorithms’ successes and failures, definitions would then be adjusted, accounting for disruptions that had not previously been detected, and letting harmless ones pass in the future. Being modifiable, definitions offered a balance of flexibility and clarity that let members and algorithms adapt to one another. Disruptions were defined in a few ways: in terms of bounds, comparisons, and categories.

Most commonly, members demarcated bounds around the typical or expected range of outputs for particular activities (See **Figure 4** for an example). Algorithms could immediately

flag outlier activities when outputs exceeded what members considered “reasonable values.” For example, “with satellite data, you expect a certain range,” an engineer noted, “and if it's outside that range, something's wrong with it... we might throw it out.” Extreme weather values and atypical crop behaviors are other examples.

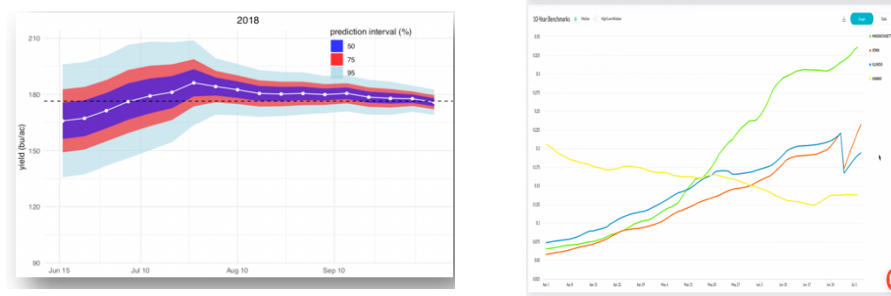
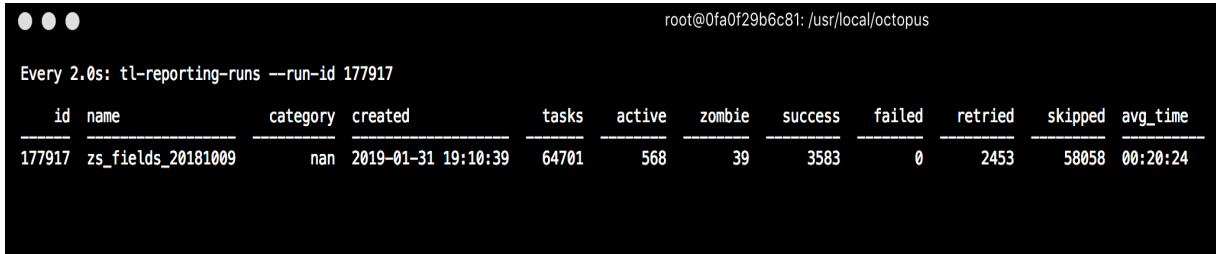


Figure 4. Example of bounds and comparisons defining disruptions

Additionally, members designated benchmarks to compare live outputs against. When it comes to weather statistics and crop metrics, historical values, as well as values from alternative sources, could be referenced for comparison. Inconsistencies and large deviations raised concerns. Several kinds of comparisons were used. One was the “side-by-side,” which visually depicted comparisons in adjacent images or charts. Another was the “difference statistic,” which displayed, in numerical form, the benchmark difference. Difference metrics applied to code as well as data, and displayed in a salient way whether lines of code changed over time (i.e., x% has changed). Comparisons enabled members to spot inconsistencies at a glance. “We should run these comparisons all the time” an engineer urged.

The use of categories to define disruptions was also important, though less common. An example of this is the categorizing of negative tests. Because outputs could be valid even if they

appeared out of bounds, or at odds with benchmark comparisons, false-negatives were important to identify (See **Figure 5** for an example).



```
root@0fa0f29b6c81: /usr/local/octopus
Every 2.0s: tl-reporting-runs --run-id 177917
```

id	name	category	created	tasks	active	zombie	success	failed	retried	skipped	avg_time
177917	zs_fields_20181009	nan	2019-01-31 19:10:39	64701	568	39	3583	0	2453	58058	00:20:24

Figure 5. Example of category definition of disruption in ‘failed’ and ‘skipped’

One engineer created his own “reporting module” algorithm for this purpose. It categorized all disruptions in a certain operation as either a “fail” or a “skip” disruption. He explained:

You have tens of thousands of tasks... Some tasks can fail for good reasons, and it's ok that they failed... sometimes it's [data] just not there, there was no [satellite] observation that day, then you should allow that. that failure should be skipped, it's ok. But sometimes, maybe it [imagery] really should have been there... for a particular kind of error that we deem acceptable, we'll raise a certain kind of exception with a special name called “skip task,” and the reporting module sees that and says, ok that's alright, I won't call that task “failed,” I'll call it “skipped”, and therefore I'll keep going, I won't try to retry it. We won't do all that other stuff we do [for failed tasks].

In order to keep up with speed and scale of operations, members generally tried to define “false failures” in advance (as ‘skips’) so that, when they arise, they do not stall operations.

If you don't skip anything, you'll be loaded with failures that, then, aren't really failures. And you'll have to personally go in and skip them, [which does] not allow the system to move fast enough. that's not desirable either.

With bounds, comparisons, and categories of disruptions defined, algorithms could continually scan ongoing operations and flag issues proactively, while members attended elsewhere. In a glance, members could notice disruptions that they themselves had defined. As one member explained, it “will be super wonky, and you can’t miss it.”

Importantly, definitions of disruptions are flexible, and can be adapted to evolving contexts. On one occasion, a member told the team that they “had bad range values for the plant health metrics,” and another engineer responded: “I think we should manually override... go ahead you want to give me a [new] range and I will apply.” By specifying definitions, and catching disruptions that needed attention, members extended their reach, gaining visibility over ongoing work. As a result, they could oversee much without inspecting everything.

Displaying disruptions enabled members to see what algorithms flagged as disruptive. Displays, generated algorithmically when definitions were breached, alerted members to potential disruptions. This depended on members and algorithms integrating programs, setting triggers for display based on definitions of disruption, and transmitting displays to those who could respond to the disruption. Engaged in their present work, members counted on these disruption-detecting algorithms to raise visual alerts when otherwise routine operations seemed disrupted. Almost all members worked in front of at least two computer monitors, and typically had open multiple windows, browser tabs, and applications. They configured algorithms to display disruptions directly on their screens. “It’s a very visual process... If you can’t see anything... you’ll never spot the error,” a member noted.

The modular design of algorithmic operations, allowing the outputs of one component to become input for another, meant that different programs could be integrated together. Members could easily peer into the activity of many operations in the same workspace, through integrated programs. It is akin to accessing many distinct webpages from the same browser, by integrating tabs, or logging into many distinct accounts from the same single-sign-on, by integrating app permissions. Through integrated programs, members could see alerts and displays from many distinct programs co-located on screen. No matter if members were writing code, manipulating

data, sending messages, or working on a presentation, live operations were not far from view, as the disruption-detecting algorithms displayed metrics and visuals before them. For example, Slack, an internal messaging program, was integrated via APIs (Application Programming Interfaces) with many other programs. Some Slack channels included bots representing other programs, who sent alerts to members when disruptions were detected. In one case, an algorithm detected an anomaly, displayed it to a member on Slack, and was then resolved within minutes.

The documented summary reported:

Our anomaly detection [tools] triggered PG alerts on 2019-08-30 at 8:30a and acknowledged by reported details to [#engineering Slack channel] at 8:46a and [a member] began initial triage at 9:10a.
[The disruption resolved] at 9:52a.

To instruct algorithms on when to display alerts, members set triggers, embedding their definitions of disruption into various sensors in the live operation. Without a trigger, algorithms would not be able to signal members of an issue, even if definitions were spelled out. Many triggers were offered as-a-service by external providers who continually ran tests on AgMap's operations (i.e., Pager Duty, DataDog, Cloudwatch, Sentry, CircleCI, Pingdom). Many were also internally developed. A basic trigger familiar to all programmers is the "if/else" conditional statement, (and "exception" statements) which could prompt algorithms to display disruptions if specific definitions were breached. An example of this occurred after a data update disrupted service. That particular update was not caught because there was no trigger for it. To avoid this particular case in the future, new if/else triggers were set. A member explained:

Temperature [data] for 2019 didn't arrive [from data source as expected], and it broke our code. The jobs I tried running all week kept failing... there just isn't a 2019 file. I'll add more if/else statements.

Using integrations and triggers, members were able to expand their field of view. They were alerted to potential disruptions without having to pry into the 'black-box.' An engineer

explained: “[It’s] a problem if you have a human to go through the 10,000 line log file... To make it cognitively sustainable... You should not have to look inside all the boxes.”

Due to the distributed nature of AgMap operations, those members who knew what to do were not always the first to spot the issue. They thus counted on others to transmit displays of disruption. For example, one engineer who did not work on zonal statistics directly, but encountered them in his front-end work, noted that he “doesn’t mind” passing along alerts:

They [back-end engineers] would have to be looking exactly at the one value that’s off, say in Argentina in 2001, and that is so improbable... I don’t go deliberately to check, I’m working on a totally separate thing, but I’ll be able to see if something is wrong.

To assist onlookers in interpreting the displays, members attached varying amounts of additional detail. Most times, they simply redirected the display, without commentary (see **Figure 6** for an example). Because digital displays were easily transmittable, people frequently copy-and-pasted or forwarded them to one another, mostly through Slack channels and task ticketing channels (digital task management tool). What they distributed were screen-shots of discrepant charts, figures, and images. Other times, they summarized what they saw, sending brief context along with the display. Summaries quickly directed attention to what was salient. For example, a member wrote on the group Slack channel, “not urgent, but there seems to be something funny with the [crop health] data... seems too brown” [with image attached].”



Figure 6. Example of disruption in zonal statistic operations shared on internal wiki

In complex cases, members offered further elaboration, adding interpretations, perspectives, and observations to make displays more meaningful. Elaborations were especially important when the possible causes of disruption were numerous or seemingly out-of-the-blue. A data scientist told me of one humorously unexpected update that affected her data. Thanks to “a feedback loop with people in the field [where data were collected],” she learned that “a bird pooped on my sensor and prevented the data from transmitting.”

Given the importance of continuous and consistent operations in live service, it was urgent that the appropriate workers be apprised quickly. This expectation was articulated one day when displays of disruption were transmitted with significant delay. A few data scientists learned that the engineers received alerts indicating that an update may have disrupted AgMap’s data but did not transmit the alert. One of the data scientists found out only weeks after the update, and demanded: “if there are entire days of missing data, we need to know immediately!” This incident resulted immediately in the formation of a recurring face-to-face meeting about data updates and a new Slack Channel for transmitting displays.

In summary, members and algorithms defined and displayed disruptions, covering an indefinite span of time and components in which updates could turn disruptive. What counted as disruptive could be reprogrammed in definitions and triggers, and transmitted through integrated programs. This enabled members to quickly catch updates upon turning disruptive, and to adapt their mechanisms indefinitely as live operations progressed, but these practices also generated a pervasive vigilance, raising alarm at any time about any component for concern that updates may or may not turn disruptive.

Restoring Functionality

Given the priority to keep live operations functioning continuously and consistently, AgMap members and algorithms responded urgently to disruptions (i.e., inconsistent data, discrepant code, or dysfunctional infrastructure from an update). But disruptions were not easily resolved during live service. For one, updates occurred unexpectedly, creating unique disruptions for which no exact fix could be prepared in advance. For another, by the time members detected the update, the service was already disrupted, leaving no time to prepare a fix then either. AgMap took an approach to resolving disruptions that members could prepare for and quickly implement regardless of the particular update. This approach, which I call *restoring functionality*, consists of activities capable of temporarily reverting operations to pre-update versions (i.e., prior versions of data, code, and infrastructural configurations that were known to function properly). Specifically, members and algorithms were *preserving functionality* preemptively, and then, once disruptions emerged, temporarily *reinstating functionality*. Before describing each sub-practice, I provide an overview of how functionality was restored in live operations, drawing on an example.

In one instance, a data improvement was incorporated that changed the upload location of a routine data stream, but the code that processed the data was not updated accordingly. Consequently, continuing to search the previous location for the data stream, the existing code led operations to ingest the wrong data and to produce zonal statistics violating previously defined bounds, thus raising a disruption. Once the disruption was detected, members immediately restored the disrupted operations to original functionality. An engineer proposed temporarily “rolling back” operations to a version from the prior day:

It looks like yesterday's clean outputs for MODIS [satellite data] had all of the dates. I think we should restore yesterday's version of the outputs... That will fix the gap but will mean we no longer have data for today but we'll have a full history to build on when we fix this.

This “patched the gap,” and the head of AgMap applauded the rollback, saying “[it’s] much better to be a day behind than to have a sort of ‘warp mode’ hop.” The view was that reinstating prior operations, as a stopgap solution, was preferable to leaving evident inconsistencies (in this case, a ‘hop’ in data plotted over time) in the live service. Not until functionality was restored were members able to introduce additional improvements.

Preserving functionality provided members with functioning components to fall back on, as needed. Before operations could be rolled back and their functionality restored, prior versions had to be available and retrievable. To prepare for this, members took care to preserve the components of service in full and specific detail as they worked. While it was possible to simply recreate a prior version when needed, at AgMap’s scale, it would cost too much time and computational resources to redo the work. As one engineer noted:

Before [we] can summarize [create zonal statistics of] anything, we must first retrieve the geometries [imagery data] to be summarized and [process] them, these preparation steps can be quite expensive.

AgMap’s zonal statistics used numerous large databases, codebases and computational systems, and took many hundreds of thousands of computations to complete, requiring use of cloud servers. For example, in one case, after six hours of processing, a few members reported “those tasks are still going... . . . still 300,000 more, or another 12 hours.” Another member wrote:

I can’t get an intermediary product until it’s all done. [It’s] such a big system. The momentum is so extreme.... I just have to be patient and let it all finish.

Facing the high expense of redoing work, and a downtime exceeding what was expected of continuous operations, AgMap re-used previously “finished” work (prior operational components) instead of redoing that work to regain function. For instance, a member documented that, “[We] cache stats from previous year so they don’t need to be recomputed on

every run.” They were able to preserve functionality by holding, stamping, and tracking their work using cloud resources and sophisticated storage techniques. Holding prior work components, like holding a library book, involved storing the components, whether for a long or brief time. Members stored not only the final outputs (i.e., zonal statistics) of prior operations, but also the versions of data and code used to make those exact outputs. The longer the time frame for retrieval, the more important it was to label and log the work carefully (i.e., the operation, date, batch), so that details would not be forgotten. A member noted:

[Because] operations are constantly updating and revising data... we store all of the versions... [so] we can reconstruct the sequence [when we need it].

Stamping was an indirect approach to storing that nevertheless preserved prior work.

Unlike holding, stamping preserved work automatically, without anyone having to remember to do so each time. It is like a book stamped by the library with property claims and detection strips, that is able to return to storage if lost or stolen. For example, code management tools, like Git, retained histories of all code ever committed, stamped by timestamp, which enabled retrieval of prior code anytime. A member said:

Even things that have been overwritten... It keeps track of evolving code, all commits ever pushed. We can easily revert back to any earlier version.

In my field notes weeks later, I noted a different member explaining in a standup meeting how stamped work would be quickly retrieved:

Now [Engineer] will look through git log and rollback to an earlier version, before the big change [a change in how data was binned] was made.

Data too were preserved by stamping. Sometimes with data, the act of deleting data did not have the effect of removing them, but of merely stamping them with an indication for removal—what members called a “soft-delete.” In that case, when members or algorithms issued an instruction

to delete data, what actually happened was that the data were marked with the intent to delete, but remained nonetheless preserved with an associated ‘delete’ indicator.

Tracking all this work in specific detail was crucial for replicability. Holding and stamping helped preserve prior content and functionality, and tracking helped retain specificity, so that operations could be rolled back accurately. In a live service, where components were updating all the time, it was easy for members to lose sight of or forget the details of what they had previously worked on. The fact that components updated concurrently and indefinitely made it all the easier to forget how components fit together. One member related:

Many things are flying... We’re working with data and getting it at the same time. I must have worked on four versions for this [algorithm] in the past, and I forget which one assumes what or uses what kind of data.

Members wrote about their persistent concern that “changes [elsewhere] can unexpectedly [affect] them,” and that without careful “tracking, suddenly we cannot replicate our zonal summary results.” This is a problem when working with algorithms because data, code, and infrastructural components are fit to specific parameters and precise configurations. Prior work was only functional if pieced back together properly, and to do that, it was helpful to have tracked specific version details, assumptions, and history of development. This helped avoid problems where, for example, a prior version of code was paired with the wrong version of data. The aim of tracking, as one engineer put it, was to be able to “always recover any changes that were made.”

Third-party services were used to track operations, and ran in the background with minimal intervention by members. One such service was Docker, a widely used code deployment tool, which provided members a way to package their algorithm components so they could be run on any cloud computer. Docker assembled the specific components required to run

operations, including requisite data, code, and infrastructural elements, and “froze” it into a “virtual environment,” as members referred to it, so prior work could be done in “exactly the same consistent environment,” no matter what computer ran it or when it was run. On one occasion, a particular code component updated unexpectedly, disrupting the rest of the operation, and Docker helped members recall the exact pre-update configuration of all relevant components in their appropriate versions. A member explained:

I want to get the old version [of an operation system] back because that's what works with the code, so... we actually had to lock those down [in Docker], because people change things and it breaks the code.

Often, there was a need to recall why something was done, who had done it, or in what context it operated, and this information might not be captured by third-party programs. Members therefore made and managed their own tools to track such details in real time. One such tool was the digital lab notebook, described by members as “a place to record important actions taken,” which they configured to keep an automated log of who made what change when. Another tool was the custom Slack channel that members repurposed to log activities. For one member, that channel was named #production_journal, and it stored specific code commands, or as he described: lines of “long and complicated” code. With these tools, members were then able to quickly recall specifics. This reduced the amount of details members had to keep top of mind.

As I have shown, algorithms had a large role in preserving service functionality, helping to hold, stamp, and track members’ work at a scale, speed, and precision beyond people’s capabilities. A good example of this is the way algorithms preserved themselves. For many algorithms, its data helped preserve its code, and its code its data. Members wrote code to compute the proper version of data to use for a specific function, and they created data (as meta-data rather than as input data) specifically to instruct and guide algorithms. There were data

tables that “keep track of the versions of the algorithms that have been run... to track the work.”

With specifics tracked, algorithms would “be able to reproduce [the] work we do,” since

Each run would know which version of each [code components] to pull down and exactly which [data] files need to be fetched to... replicate the state of the [operations] that were run.

The benefit of preserving functionality was the readily accessible and immediately usable base of data and code that members could build on when time was limited. With prior functionality ready at hand, members could resolve disruptions relatively quickly, substituting them in without stopping service. My field notes have multiple references to members’ concerns about “ensuring that we can fallback,” that if a certain interaction “stops tomorrow, we can continue functioning” and “recover from an... outage [update].”

Reinstating functionality brought those preserved functions into live operations. With prior functions preserved, members could put them to quick use, reinstating functionality as soon as disruptions were detected. Reinstating functionality helped resolve disruptions temporarily because, with prior versions substituting out the disrupted version, operations could regain consistency. However, for prior versions to function properly, they had to fit together. If prior data, code, or infrastructural components did not match, such that benchmarks, parameters, or configurations were inconsistent, function may be reinstated but still display disruption.

I found two approaches for reinstating functionality to ensure consistency, one which involved fully rolling back a component to a prior version, and the other which was more about partially blending different versions. Full rollback emphasized reverting not just one line of code or some subset of data, but all relevant code, data, and infrastructural components to reproduce specific functionality. One example of this is the code script a member wrote called “restore-yesterday.py.” Once executed, this script quickly reinstated “yesterday’s” operative conditions,

effectively “rolling back” operations to a familiar version with all the specific component functionality corresponding to that version. Another member commented on the speed benefit of the rollback script, which rendered the prior functionality automatically: “without this [script]... resolving [the disruption] would have taken much longer and manual pre-fix revert would have been impossible.”

Although it was sometimes crucial to reproduce prior functionality fully, such as when operations were completely invalid, on most occasions, disrupted operations could still function (i.e., produce outputs) even if inaccurately or inconsistently. In these cases, partial blends were preferred instead of full rollbacks, so the reinstated function would not break continuity of service. The full rollback was required when, for instance, data were completely missing due to a delivery update. But repeating prior outputs fully in this way indicated a static service. Partial blends were desired when, instead, data values were merely transformed by a parameter update. In that case, members opted for a blend, mixing present and prior to fade out inconsistencies. Said differently, the aim of blending was not to replace updated versions with priors completely, but to have the new sufficiently weighted by the prior. For example, with statistical outputs, various compilation techniques like averaging, splining, and compositing mixed prior content with updated values in a blended form that appeared normal, or at least more consistent, to users. In one case, when a certain production of zonal statistics appeared discrepant, members immediately began “filling the missing gaps with long term mean values.” A member explained that, generally,

If there's problematic data we tend to either completely throw it out or infill it with medians of other existing data we feel more comfortable with.

On a different day, the head of engineering, in response to a recently detected disruption where the outputs statistics appeared discrepant, articulated this infilling approach, saying:

The charts will be a little rough. We need an option for a spline [smoothing line]... just because it's going to look bad... it looks jagged.

Workers rarely had to cease operations on account of a disruption because priors could substitute on short notice. This was important, “we should never be blocked by anybody else,” an engineer once commented. Prior work served usefully as “place-holders” while workers developed improvements. In one case, when a data component updated unexpectedly, members immediately presented users with a mash of prior work, thus buying themselves time to properly address the disruptions. An engineer explained:

We... just smashed them together to have data we wanted. As a placeholder ... The source we were waiting for has matured and so soon we will swap it in for our compiled... dataset.

In summary, members and algorithms preserved and reinstated functionality, allowing for almost immediate recovery of a live service when updates turned disruptive. Because there was no way to predict what components would need recovering or when, members deemed it crucial to track, store, and maintain in specific detail all of their work indefinitely, or until the cost of doing so did not seem justified. This enabled members to regain consistency and continuity of service quickly when disrupted, but these practices brought forth outdated components. Additionally, in revisiting prior work, members scrutinized and explored additional improvements with the potential for new disruptions.

Consequences of Coordination Practices

The three coordination practices—incorporating improvements, detecting disruptions, and restoring functionality—helped AgMap temporarily achieve continuity and consistency in an ever-improving live service (see **Table 4**). Because improvements were introduced through heterogeneous, concurrent, and indefinite updates, improvements occasioned disruptions. Disruptions, in turn, galvanized efforts to revisit prior work, restore functionality quickly, and

get operations running again. When the conditions of work shifted, and live (consistent and continuous) operations became disrupted, workers gained temporary advantages from the urgency of disruptions, and this helped AgMap *stay live through disruptions*. The drawback of working in a disrupted state, though observable in each incident of disruption, stood out more prominently when viewed as an ongoing stream of recurring disruptions. As I saw conditions of work shift frequently and systematically into and out of disruption, I began to understand that the practices I identified enacted a different approach to coordination than suggested in the literature, one that, by leveraging disruptions, helped AgMap *grow a live service by perpetuating disruptions*. I present the consequences of working through disruptions and furthermore, through perpetual disruptions.

Practices	Challenges (why)	Sub-practices (how)	Consequences (so what)
Incorporating Improvements	Systems are incomplete and constantly changing Concurrent use and development reduces time to align on updates	Initiating improvements with relative independence and autonomy Introducing improvements into live operations with documentation	Address collective goals through specific and prioritized tasks Make updates to live operations without extensive deliberation
Detecting disruptions	Components update invisibly and unexpectedly, disrupting operations Updates elude members' notice and recognition	Defining disruptions with bounds, comparisons, and categories Displaying disruptions by integrating and triggering	Identify emergent disruptions precisely, flexibly, in real-time Notice disruptions immediately with visible local cues
Restoring functionality	Operations can fail unexpectedly and disrupt live operations Disruptions impair consistency and continuity of service	Preserving functions at scale by tracking, holding, stamping Reinstating functions in context with full rollbacks and partial blends	Access and retrieve prior, functioning versions Resume operations temporarily with functioning components

Table 4. Regenerative coordination practices and sub-practices

Staying Live Through Disruptions. Updates are neither de facto improvements nor disruptions, but can, as I have shown, originate as one and manifest as the other. When updates turned disruptive, ways of working changed accordingly, to quickly turn out improvements. These changes challenged workers, as they pushed workers to do more in less time.

Firstly, resolving disruptions took priority over other tasks, as disruptive updates compromised the continuity and consistency central to live service, which meant workers had to be able to drop whatever they were currently doing, or discard whatever they were previously working on, to address an emerging and urgent disruption. Members referred to moments of disruption as “being on fire,” and called them out in Slack chats, meetings, lunch conversations, and interviews with me. One member once joked, pointing to a colleague who was dealing with the disruptive effects of an unexpected update, that he had just “run into a fire... and it burned off all his hair,” having worked on resolving it the entire night before and all that morning. The onset of disruptions delayed ongoing work and added to the “backlog” of tasks members juggled. An engineer told me one of his biggest frustrations was having to always postpone work for disruptions, adding work to the “backburner,” and being unable to resume it for a while. Another frustration I frequently observed occurred when members had to discard or rework previous improvements that took a lot of effort to “complete.” In one case, an engineer took pains updating code to improve the color scheme for imagery data. Months later, he was pressured by the team to redo it when imagery data updated and showed unprecedented moisture levels, distorting the color scheme. He fought back: “So I disagree with both of you because I spent weeks tweaking this color map and I believe it is as good as it is going to get for a while,” but in the end, to resolve the inconsistency it caused, he rewrote the code again. The practices described in this study gave members and algorithms a degree of adaptability, helping them work through emergent disruptions, but working in this distributed way interfered with members’ ability to plan, schedule, and commit to specific workstreams.

Secondly, because disruptions resulted from the indefinite updating of algorithm components, which followed no specified time frame, workers had no way to know when a

disruption would emerge, or how long it would take to resolve. Once a disruption emerged, it claimed priority, and members were caught up in it. By detecting disruptions and restoring functionality, AgMap could display and respond to disruptions quickly, potentially at anytime for any component, and this accelerated the pace of work. But working in a disruptive state of operations was also incredibly stressful and exhausting. Disruptions that emerged at the end of the work day, on weekends, and in the middle of the night, compelled members to respond at off hours, to cancel personal plans last minute, and to scramble for responses. I found in archival materials a document listing personal phone numbers which encouraged AgMap members to call one another at any time in case of disruptions, and found in the Slack records many instances of alerts sent by algorithmic bots to members timestamped after midnight. In one particularly emotional encounter, I talked to an engineer who had “not been able to sleep freely” “worrying about things” after dealing with a series of disruptions (i.e., the cloud servers where he ran AgMap operations kept updating the computers his operations were assigned to, thus terminating his operations unexpectedly each time; and the NASA repository where he got live imagery updated its data stream when the government shutdown and again when one of the satellites glitched). He told me that he was freaking out, almost in tears, because “everything is failing and I can’t figure out why or where.” He added, “What a waste of a day... I might as well have just stayed home.”

Thirdly, the autonomy and independence that workers demonstrated, which supported individual initiative and facilitated quick responses to disruptions, were also isolating in several respects. As members initiated updates concurrently, and dealt with disruptions in their own sandboxes, members seldom deliberated with others about the details of their updates, and only did so after they had already done much of the work. This preference for retrospective

deliberation, in lieu of ongoing alignment, delayed communication but sped up progress when members were confident about what to do. Issues arose in moments when understanding was needed but lacking, such as when the person responsible for sustaining a component was tied up, out sick, or on vacation, and when the relevant algorithms were broken or offline. An engineer expressed the frustration this caused once when a disruption emerged from a code update just as the colleague familiar with it was out:

I'm completely stuck on this. I have no idea how to proceed. I just don't know what to do. I know at which stage the errors pop up, but I have no idea how to address it. This isn't my code.

Another issue with autonomous updating was that members felt that a lot of the work they did, especially when responding to disruptions, was unrecognized or underappreciated. Because their task management boards listed ongoing assignments, unexpected disruptions did not show up right away. I had often observed members working the better part of a day resolving a disruption that was not officially accounted for. One data scientist said in a team meeting “no one knows [about the task I did] except me, it just doesn't show up anywhere.” Members mentioned these sentiments when talking more generally about “technical debt,” which referred to aspects of algorithmic data, code, and infrastructure that were susceptible to disruption, for which additional improvement could make more reliable but was not currently necessary for operations. A lot of effort went into reducing technical debt, yet this tended not to be noticed by those unfamiliar with the particular component, who only used it as a “black-box.” More than a year into my research, management addressed this in an all-hands meeting, where a slide was titled “overemphasize wins,” adding “even calling out a perfect execution on something that you normally wouldn't call out as a win can go a long way.’

Finally, the priority for workers—keep operations consistent and continuous in function—encouraged quick progress, but also shifted the objective of work. In moments of disruption, rather than ensure long-term viability of immediate work, members responded to disruptions with the aim of getting operations going again, which usually meant do just enough to keep operations live. A data scientist articulated this sentiment after making a compromise to keep users satisfied:

So I end up with a three-legged duck, where I have half of what I need and then I can prove half of what they want to see, and...that's not really satisfying, but still enough to move forward.

Members talked about disruptions as exigencies caused by actors beyond their control, which they were responsible for addressing but not necessarily responsible for causing. After one disruption, involving a code update with no corresponding update to the cloud computers running that code, an engineer said of such disruptions “it’s inevitable... there will always be an error. Happens even to [head engineer]... so this is not something I did.” He emphasized that his objective was to resolve the issue now, not to eliminate the possibility of it occurring again. In another instance of disruption, an engineer explained that it was no one’s fault, not a sign of negligence or personal failure, but as a normal feature of a live updating operation, for which the proper course of action was to quickly reinstate the operation and proceed with a fix.

We try to handle known errors, but we did not anticipate this... the job failed, but there’s no failure for me [we can deal it with now because it is an issue now]... There’s always a problem... good projects will have them.

My interviews with other engineers similarly found that they oriented their objectives toward resolving disruptions, and in some sense disassociated personal responsibility for causing disruptions. One engineer explained:

You need thick skin. You have to be disconnected from the results. So many things... that could have gone wrong. It's not going to ruin my day. I'm not going to let it ruin my day. I'm responsible for it, but it's not me.

My findings suggest that AgMap did not treat disruptive updates as faults to be eliminated, nor did they view disruptions as problematic incidents, so long as they could be resolved quickly. Furthermore, my data suggests that the coordination practices that helped AgMap stay live through disruptions, also helped AgMap grow the live service through disruptions. Disruptions were important, not only as occasions to recover from, but also, as I show next, as conditions to accelerate improvements.

Growing a Live Service by Perpetuating Disruptions. Working in a disruptive state of operations was advantageous (and challenging) for the reasons previously described, and one way of leveraging those advantages was to perpetuate disruptions (and its consequences). There were two primary mechanisms for perpetuating disruptions. The first was through incomplete updates, which, occurred when workers left components updated sufficiently to improve operations for the moment but still capable of turning disruptive anytime operations updated further. Think of incomplete updates as seeding disruptions. The second was through restorative updates, which occurred when workers resolved disruptions with new improvements going beyond the minimal needed to restore disrupted functionality, thus initiating a new round improvements, disruptions, and repairs. Think of restorative updates as precipitating disruptions.

Incomplete updates were updates intended to improve a component, and which indeed improved functionality, but were also likely to turn disruptive in future operating conditions. Of course, disruptions were to some extent unavoidable given the heterogenous, concurrent, and indefinite updates that live service experienced, but what I found was that many disruptions were seeded in intentionally incomplete updates. Components were usually updated to a state

knowingly susceptible to disruption from future updates. Members felt that the work required to fully address vulnerabilities took too much time, or that, in any case, update disruptions were unavoidable given the nature of live service, and it would not be worth the time to attempt eliminating. Rather, they encouraged a kind of work that, in interviews, meeting notes, and wiki docs, was described as “fragile,” “hard coded,” “delicate,” “spaghetti,” and full of “technical debt.” An engineer once admitted candidly: “I’m not working it [a code component] at all right now, even though [I] know there are issues. I’ll come back to fix those [vulnerabilities] later.” Another engineer told me: “I don’t aim for perfection... I don’t even aim for it to work... Even if it works, it’s not finished.”

The aim when developing an update was typically just to get operations to work “for now.” That allowed members to quickly grow the service with new developments. In one document, members shared their thinking on a particular code component in development: “[We should] fill only those events required for [current needs]. We can chase completeness later if there is a use case.” AgMap even disclosed their update incompleteness to users, in reports, presentations, and dashboards attached with the explicit disclaimer: “These materials contain preliminary information that is subject to change and that is not intended to be complete.”

Members spoke about leveraging incompleteness to accelerate work. Completion, to the extent it was even possible, was seldom a priority, and usually compromised, in favor of consistency and continuity. Members felt comfortable seeding disruptions through incomplete updates in part because they were already coordinating work in a way that could quickly detect and restore disruptions. Knowing that they could quickly reinstate functionality when and if something went wrong, members felt confident perpetuating disruptions. Once, when an engineer proposed extensive tests for a forthcoming update, a colleague recommended: “My

advice is skip it [tests], and worst case we can just revert it,” implying that the update testing wasn’t worth the time to try to “complete” since, if the update somehow turned disruptive, it would be quick to restore.

Restorative updates were updates intended to restore functionality, and which indeed kept live operations functional, but were accompanied by additional improvements which were not necessitated by any existing disruption. What a disruption allowed, thanks to its urgency and the priority it justified, was a gathering of attention on prior work, as members and algorithms inspected and explored existing operations in their attempt to detect and restore. I found that members took advantage of this opportunity, initiating and introducing extra improvements, going through “backburner” tasks, newly identified vulnerabilities, and update opportunities.

As members reinstated functionality, reexamining prior data and prior code to remedy a particular disruption, they noticed additional vulnerabilities, and raised to-dos not otherwise on their minds or on the agenda to work on. In one instance, a data scientist was restoring a disruptive data update, and in the process realized that other updates should be discussed. He wrote to a colleague in Slack:

This will probably be solved by current work you are doing... but it would be good to discuss to make sure... I don't disagree but we need to align with you so that current work we are doing right now lands where it needs to.

In another case, a different data scientist, upon hearing of a recent update, prompted discussion with colleagues about the need for further updating: “The way they are using the weather data now is not the way anyone else would do it... we need to have a conversation.”

My findings suggest that the time justified for restoring urgent disruptions was leveraged to incorporate improvements. Another example of this was when members suggested that a recently updated code component, once restored, be applied to other parts of the service. The

code was changed in response to the unprecedented Midwest floods in 2019 (an anomaly by mother nature which turned an update disruptive), and improved the accuracy of detecting water in satellite imagery. The director of AgMap saw it and commented on Slack:

FWIW [for what it's worth] my instinct is that a lot of this capability described above will be part of "normal business" even in years that are less weird than US 2019.

This led to the consideration of new work, reaching beyond the scope of the original disruption. Not only would the current disruption be resolved with an update, but additional updates would be considered (i.e., using different data altogether, or switching to different code altogether). A data scientist captured this idea well:

There may be [additional] asks [for improvements]... it's not brand new, it's stuff we're working on, but new manifestations [of it]. [People] have a new shiny object now, and will likely push my requests in that direction.

In summary, because disruptions were in some respects unavoidable, AgMap coordinated work through unique practices to keep live operations functioning continuously and consistently—by incorporating improvements, detecting disruptions, and restoring functionality. Yet, not only was AgMap able to recover quickly from disruptions, but the same practices led AgMap to perpetuate disruptions further, leveraging the urgency of working through disruption to quickly grow the service (i.e., adding to prior data and code). Two points are worth emphasizing. First, the very act of resolving disruptions led to updates with the potential of turning disruptive (such as through incomplete updates and restorative updates). Second, as updates could occur over an indefinite timeframe, the guiding priority for workers was to sustain consistency and continuity of operations over the life of the service, as opposed to ensuring completeness or correctness at any particular moment. As a consequence, workers perpetuated the stress that attends this indefinitely disruptive and open-ended work. Written in the meeting

notes of a team retrospective, under the header: “what we could have done better,” the desire for some sort of closure was made explicit: “we need a clear definition of what “done” means.”

Discussion

This study sheds light on a distinctive way of coordinating work now prevalent among live, cloud-based platform services. A live service is developed and delivered through algorithms whose components undergo heterogenous, concurrent, and indefinite updates. Coordinating the work required to manage the disruptions that emerge in such conditions entails three distinct practices. I propose a model of regenerative coordination where these practices accomplish the ongoing regeneration of the service. The model highlights how members and algorithms keep a service live and growing by resolving disruptions through updates, which paradoxically ends up perpetuating disruptions (see **Figure 7**). I articulate the insights that a perspective of regenerative coordination offers, and discuss how it contributes to our understanding of coordination in an era of algorithmic, cloud-based services.

Regenerative Model of Coordination

Regenerative coordination is collective work that “regenerates” a service by keeping existing functionality operating continuously and consistently while it develops over time. Assessed over the life of a service, continuity is achieved if the service remains operational in relation to ongoing updates, and consistency is achieved if existing functionality remains valid in relation to updated functionality. Regenerative coordination thus avoids stopping a service (temporarily or permanently) or modifying its functionality in ways no longer compatible with existing uses. The objective of regenerative coordination is to sustain a live and growing service, in which operations remain reliable and content stays up-to-date. This objective is achieved

through work practices that recurrently detect disruptions, restore functionality, and incorporate improvements.

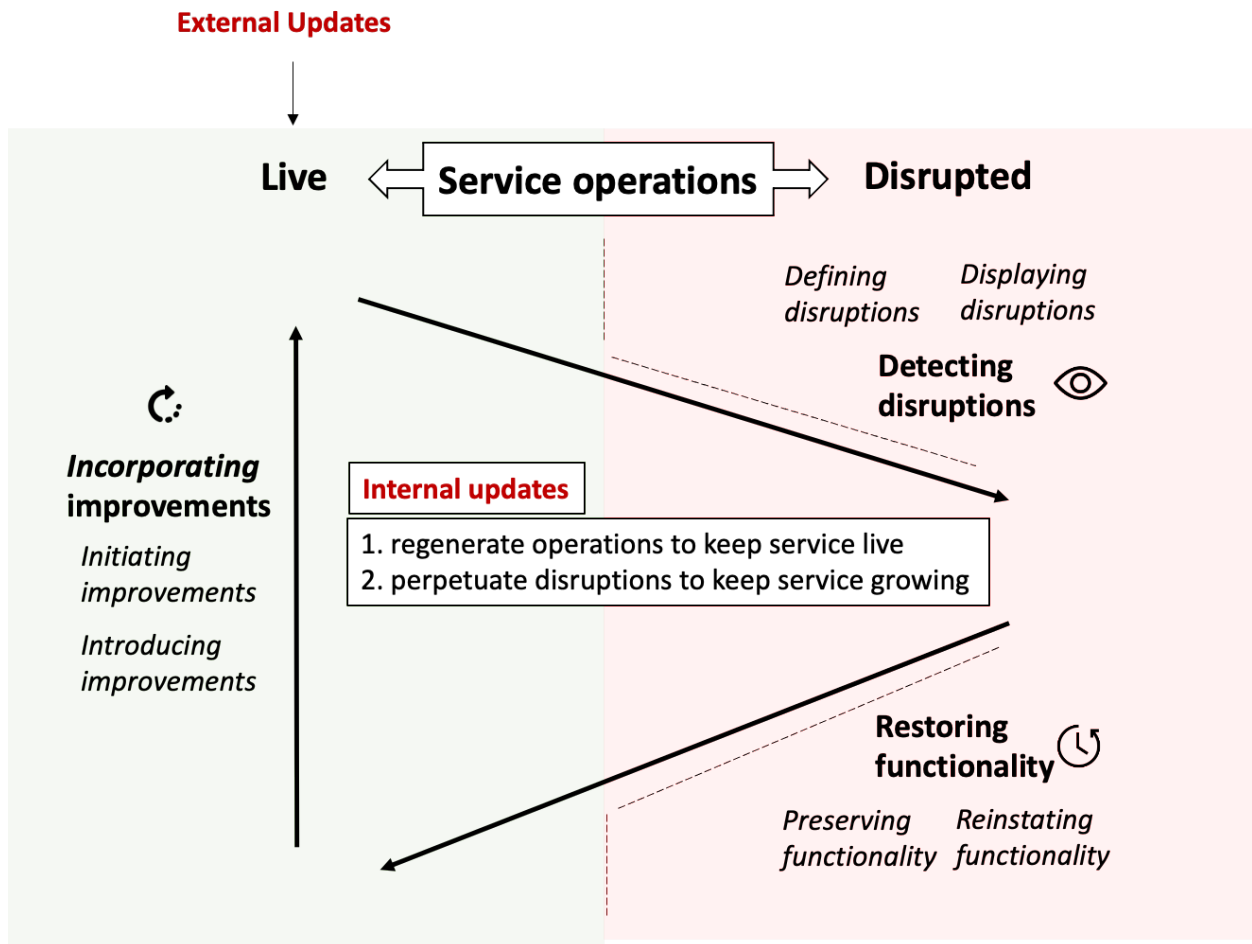


Figure 7. Process model of regenerative coordination

The model of regenerative coordination highlights the central role of disruptions in two reinforcing processes (See **Figure 8** for an abstracted depiction of these processes). First, disruptions, once detected, draw urgent attention to particular components of service in need of work, compelling workers (members and algorithms) to restore and improve them, and thus to regenerate operations for ongoing use. At the same time, when regenerating operations, new updates are introduced for ongoing development, which seed and precipitate future disruptions.

As a result, this second process perpetuates disruptions, feeding the regenerative practices of the first. The two processes sustain and reinforce one another, as workers repeatedly restore disruptions and improve service, again and again. In each iteration, the live service grows further, as workers deal with the paradoxical consequences of working in a perpetually disruptive state.

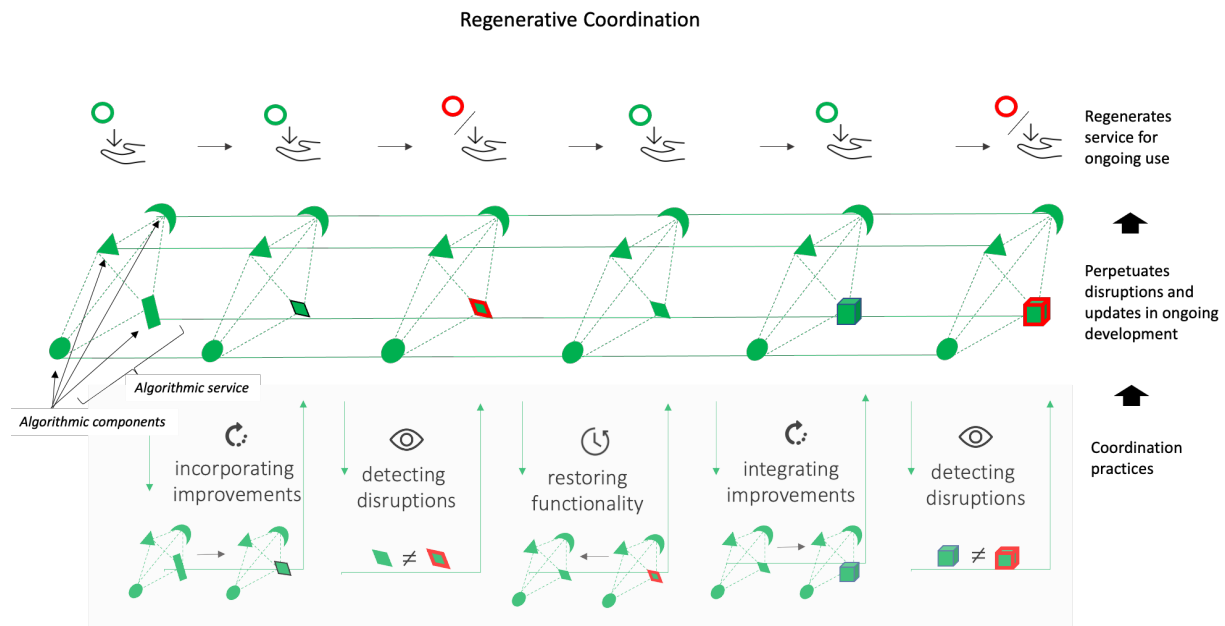


Figure 8. Illustration of live service experiencing updates, disruptions, and regeneration

A perspective on regenerative coordination offers several insights into the nature of a live algorithmic service and how the work to deliver these is coordinated across workers (members and algorithms). First, it shines light on the heterogeneous constitution of algorithmic services, and recognizes that updates enacting a live service are distributed and entangled across many algorithmic components (Kwan, 2016) and many update contributors (Kitchin, 2017). Such heterogeneity is not adequately captured by existing research, which tends to study algorithm services as singular entities producing consistent effects attributable to that entity. For instance, recent studies analyzing the effects of platform-mediated services and algorithmic work, have

attributed important issues (i.e., breakdowns, biases) to the responsibility of particular platform providers, such as “AllDone” in Shestakofsky (2017) or “Talent Finder” in Rahman (2021), or to the performance of particular algorithmic services, such as “predictive algorithms” (Christin, 2017a) or “apparatus of evaluation” (Orlikowski & Scott, 2014). This study reveals the numerous ongoing developments and disruptions that constitute algorithmic services, and finds that the services’ functionality and disruptions are, in large part, accomplished by practices that update components.

As we saw in DigitalAg, AgMap’s zonal statistic service was updated concurrently through the code packages, data feeds, and infrastructural services of many contributors working to some extent independently of one another. But these components’ functionality depended on other components. Whether each component continued to function as intended depended on whether other components had been updated appropriately. Changes in a live algorithmic service did not derive from a single entity but from the concurrent and heterogeneous updates of many actors (algorithmic components and organizational members), and this “plug-and-produce” feature of algorithms is increasingly the selling point of many live services (Lusch & Nambisan, 2015). My findings suggest that a live algorithmic service is better understood through recognizing the ongoing updates of many actors through the cloud, rather than by analyzing changes as single-sourced. I suggest that heterogeneous updating conditions make disruptions an unavoidable occurrence, not solely attributable to a particular organization or algorithm. This insight raises questions about how workers might differently enact the disruptions arising in their operations, such as leveraging disruptions to accelerate development or to justify independent initiatives as I found.

Second, a regenerative coordination perspective reminds us of the indefinite time frame over which a live algorithmic service is used and developed. It suggests that to coordinate effectively, workers have to look beyond the momentary states of a service, and act in ways that accommodate the ongoing, “always-online” objective of algorithmic services. When continuity and consistency over the life of service is the goal, completion and correctness of outputs at any particular moment may be less important. In such cases, workers are influenced not only by interacting with the algorithms directly but also, importantly, when anticipating interactions to come (i.e., vigilant about insulating, detecting, and preserving activity). While prior research highlights how people coordinate during certain moments of a live service, such as during platform breakdowns (Karunakaran, 2021), when algorithms are introduced (Pachidi et al., 2020), or at particular moments of use (Christin, 2017b), there has been limited analysis of how one moment influences actions that give rise to another. Yet, such a dynamic perspective of algorithms in use has been called for (Glaser et al., 2021; Kitchin, 2017). This study shows how particular moments of disruption, though unexpected and unintended in each instance, result from intentional actions to spur development over the life of service.

Additionally, I find worker objectives, responsibilities, and interactions supporting an indefinite timeframe for ‘completing’ development. Specifically, I find workers initiating work independently and communicating retrospectively, opting for technical debt and chasing completeness down the road, and responding to specific disruptions while keeping goals open-ended. The insight here is that workers may deliberately sustain incompleteness in their work to keep a live service going, despite the consequences of having always more to do. The notion of incompleteness has been discussed before, as a characteristic of mindsets (Savitsky et al., 1997), information (Busco & Quattrone, 2018; Jordan & Messner, 2012) and organizations (Garud et

al., 2008; Neff & Stark, 2004). In this study, I show how incompleteness is a process enacted through practices of regenerative coordination.

Third, this perspective highlights the role of mid-stream users of algorithmic services as actively intervening in the algorithms delivering service. Much of the current literature on algorithmic services focuses on end-users, thus overlooking the important influence of mid-stream users who further develop algorithms for downstream use. The literature's interest in how end-users interact with (Švelch, 2019), work with (Curchod et al., 2019), and work around (Christin, 2018) the outputs that algorithmic services provide misses the multiple ways in which mid-stream users develop, modify, or update algorithms further. For instance, Karunakaran's (2021) research distinguishes "providers" from "customers," and does not account for the customers' own development practices that customize or augment the service delivered to other users, including members of their own organization. This limits our understanding of how algorithmic functionality actually manifests in practice over time, given that other users may enact algorithms differently from the providers' original intention (Orlikowski & Scott, 2015). In this study for instance, I find workers using data, code, and infrastructure from many providers, and updating them further in their pixel factory, improving functionality, inserting detectors and trackers, as well as integrating them into Slack and other programs. This finding illustrates Seaver et al.'s (2019) point that algorithmic systems are most often "works of collective authorship, made, maintained, and revised by many people with different goals at different times" (p. 10). This suggests that users may enact an existing algorithmic service to work differently by integrating disruptive updates, and that users may view such disruptions as improvements, if they understand that updates and disruptions are two sides of the same coin.

Fourth, the tendency to perpetuate disruptions when regenerating service directs our attention to the unrecognized efforts of sustaining work (Star & Strauss, 1999). Workers' efforts in keeping a service live may be underrecognized for reasons not previously addressed in the organization literature. While a growing number of studies spotlight the stress, loss of control, and precarity associated with algorithm-mediated work, they primarily underscore issues of employment and professional discretion. For instance, studies of workers using algorithmic services typically focus on the opaque systems of evaluation on labor platforms for "gig" work (Gray & Suri, 2019; Rahman, 2021) or on the opaque decision support systems professionals use (Lebovitz et al., 2021). But as my data indicates, even skilled, fully-employed developers who understand the algorithms they use, feel highly stressed, unrecognized in their efforts, and lacking control. I point to a different cause for these consequences: where work to update and improve a service ends up perpetuating disruptions in the work. Workers may understand sufficiently what algorithms do at one moment and yet be at a loss in the next as distributed updates introduce complex and inadvertent disruptions. Workers' attempts to keep up with the indefinite stream of updates, such as by following documentations and blogs, storing, tracking, and monitoring their activities vigilantly, keeps them up at night and overflows their backlog of tasks. While workers struggle to keep a service live and always "plugged in," they may find no end in sight for their work either. As more work comes to involve algorithms on the cloud, scholars should be concerned not only about the opacity of algorithms, but also about the incompleteness generated through these updating conditions. The indeterminacy of regenerative coordination differs from provisional settlements (Kaplan & Orlikowski, 2013), because rather than assenting to a temporary close, it provokes workers to chase an indefinite end. As cloud-based services become more widespread, the understanding of regenerative practices and the

consequences of perpetuating disruptions produced in this study, may be generalizable to other settings involving the coordination of work to sustain live algorithms.

Contribution to Coordination Literature

Recent studies of coordination have concentrated on developing theories reflecting the dynamic and distributed conditions in which workers coordinate work across many settings (Beane & Orlikowski, 2015; Hinds & Bailey, 2003). For traditional settings of work, coordination models generally highlight the importance of specifying timeframes, which enables workers to anticipate distant activities (Harrison & Rouse, 2014); of utilizing local cues, which helps workers adapt to changes in activity (Kellogg et al., 2006); and of stipulating end states or output objectives, which gives workers a shared aim when adapting activities (Faraj & Xiao, 2006). Such models do not apply in the settings involving a live algorithmic service, which updates heterogeneously, concurrently, and indefinitely through the cloud.

An assumption in existing theories of coordination is that even if workers cannot collectively interact at all times they can at least interact within certain specifiable time frames. For instance, Harrison & Rouse's (2014) model of elastic coordination depicts workers interacting in cycles of alignment, coming together to voice collective issues, moving apart to work on issues autonomously, and setting up times for each of these stages that the authors label “integrating” and “de-integrating” periods. The conditions I studied did not provide workers with specifiable timeframes for interacting. Algorithmic components could be updated at any time from any number of sources for any kind of purpose, and, being provisioned live through the cloud, caught workers unawares. When updates produced disruptions, workers were expected to drop their current tasks and prioritize responding. As my findings suggest, the impromptu nature of updates offered workers no specifiable timeframe in which to interact, and led instead to a

perpetual state of relatively insulated updating, safeguarded by algorithms detecting and restoring issues, and communicated through collective documentation which could produce (but not always) retrospective deliberations after updates had been made. The capacity to be continually open to and ready to resolve emergent disruptions has not been much studied in the coordination literature. Future research can examine how workers cope (or perhaps thrive) with the ever-present possibility of updates arising unexpectedly and requiring immediate attention.

Another mechanism identified in existing theories of coordination is the use of local cues when adapting to changes in activity. In some work settings, such as firefighting (Geiger et al., 2020) and clinical care (LeBaron et al., 2016), workers find multiple cues manifesting around them, and can respond appropriately. In other work settings, such as consulting (Kellogg et al., 2006), workers curate cues for others, displaying activities through calendars, emails, and shared folders. Such a mechanism is especially helpful for cross-boundary work, where workers lacking the time or shared understanding to interact as a collective can still coordinate effectively by drawing on the multiple cues. The use of local cues, however, is complicated in conditions where the collective contributors updating an algorithmic service are highly heterogeneous, relatively insulated, working quickly, and unaware of what cues others might want or even how to get these to them. My findings suggest that workers in such conditions need to spend considerable time defining, detecting, finding, and transmitting cues independently. However, this raises a question about how workers decide whether they are producing and consuming too many cues. AgMap workers were caught in a precarious position, wanting to stay on top of all updates to be vigilant about disruptions, and wanting to push through disruptions. They experienced both stress and conflict alternating between moments of checking and flagging more updates than they had time and capacity to follow, and moments of silencing and skipping checks which then led to

disruptions. How workers in these update conditions manage to stay alert to otherwise invisible updates, without becoming overwhelmed, is an important question for coordination since live-updating is becoming increasingly prevalent in how services are being delivered today. Further research in this direction would offer insights into understanding the coordination of work in increasingly distributed and live contemporary services.

A few models of coordination emphasize the role of dialogic interactions to achieve certain end states or output goals, even if those ends are provisional. Faraj and Xiao's (2006) model of dialogic coordination in fast-response settings illustrates the importance of clinical providers deliberating their approach to care, with the end aim of discharging patients. Kellogg et al.'s (2006) trading zone model similarly addresses workers who either have specific deliverable goals or specified deadlines for delivery. But in AgMap's case, the open-ended goal of continual and consistent improvement and the indefinite timeframe for service continuance, left workers with little "clear sense of what done means." For them, updates were extensible, and could be extended indefinitely. Without recognizable markers of completion or correctness, workers were uncertain how long or how many times it was appropriate to update the same component for the same users; as a result, they ended up perpetually responding to and making updates. Unlike in medical settings, where the consequences of incompleteness, say of surgical work (Sergeeva et al., 2020), are dire, many settings where live cloud-based services are now used (e.g., analytics and entertainment services), are more concerned about losing continuity of service (i.e., downtime) than ensuring complete functionality at any particular moment (Acker & Beaton, 2016). Future research can explore how coordination practices differ from the present findings when workers need to regenerate a live service where the penalty for incompleteness is severe.

This study highlights a consequential difference between existing forms of coordination and coordination with algorithms. Algorithms update with great speed and reach, and can change operations faster than workers adapt, creating lags in coordination (Shestakofsky, 2017). This results from two now-typical aspects of contemporary algorithms. First, modular development of data, code, and infrastructural components allows workers, through component updates, to update algorithms without understanding all the details of the algorithms' components or what users do with them (Bartholet et al., 2004). Second, cloud deployment of algorithm components allows updates to manifest near-instantaneously in the midst of current operations in all the algorithms that use them (Vis, 2013). These two aspects have led many organizations to combine aspects of development and operations (sometimes called "devops"), with the hope of speeding up improvements and reducing downtime through continuous updating of live service (Humble & Farley, 2010). Moreover, algorithms on the cloud can reach heterogenous contributors, and stream updates from many components and members of many organizations, without the majority even recognizing which end-services their updates affect (Dourish, 2016).

My findings support Seaver's (2013) claim that "algorithmic systems are not standalone little boxes, but massive, networked ones with hundreds of hands reaching into them, tweaking and tuning, swapping out parts and experimenting with new arrangements" (p. 10). The updating of a live algorithmic service complicates traditional models of cross-boundary coordination. For instance, Kellogg et al.'s (2006) trading zone is helpful when members recognize who and what they are coordinating with, and have sufficient time to check for updates. But the trading zone may be less useful in keeping an algorithmic service live if it gathers primarily retrospective notes, traces of activity, and ad-hoc alerts, which workers may not recognize as urgent or salient quickly enough to use. I find that workers need to enact safeguards to mitigate the unexpected

disruptions that inevitably interrupt them. My findings highlight that in live algorithmic settings, the purpose of coordination has shifted from ensuring completion of specific objectives, to ensuring consistency and continuity of indefinite improvements, and that this is enacted by perpetuating and resolving disruptions, with little respite for workers no matter how hard they work (see **Table 5**).

When live operations become disrupted	When the work perpetuates disruptions	Consequence for workers
Restoring the disruption takes priority	Backlog of work accumulates behind prioritized disruptions	Supports adaptability but creates precarity around planning and scheduling work
Objective is to resume consistent function	Goals become open-ended, prioritizing continuity of improvement over completeness, and consistency of outputs over correctness	Enables quick progress but reduces long-term viability of work
Time frames for restoring are unspecified	Systems are prepared to restore function as soon as possible whenever disruptions arise	Accelerates pace of work and increases vigilance, but risks over-work and indefinite urgency
Autonomy of response is expected	Individuals look ahead to anticipate future disruptions, initiating repairs and improvements	Supports independent initiatives, but postpones collective deliberation and recognition of efforts

Table 5. Consequences of working when live operations become disrupted

Limitations of the Research

My study was exploratory and centered on data scientists and software programmers working on a single platform (AgMap), and within it, on a single live service (zonal statistics). Even though their day-to-day activities required considerable coding and data manipulation knowledge, most members acquired their specific expertise on the job. One member, an engineer by title but geographer by training, told me: “I’m not a software engineer. This is not my profession, I never did any of that.” The use of non-engineers to do development work is neither unique to DigitalAg nor specific to startups, as many organizations now build on open-source code, publicly available data, and commercial cloud infrastructure, provided by third party

vendors, to develop their own services. Although I did not observe the third-party developers work, neither did AgMap members. They had fairly independent control over algorithmic components, and used them with little to no communication with the providers, a separation common of modular algorithms. However, I believe that my central finding — that a service is contingent on continually updating operations enacted in regenerative coordination practices that regenerate the service and perpetuate disruptions — may usefully inform future research. To the extent that digital services are distributed through a growing number of algorithmic components, and algorithms are increasingly being deployed through the cloud, then the findings and implications of this study for coordinating work in conditions of concurrent, heterogeneous, and indefinite updating should be useful. Additionally, I believe that the regenerative practices identified here may also play a significant role in the coordination of non-digital work, especially in open-ended work (e.g., writing) where components (e.g., ideas, concepts) can be redefined indefinitely by multiple contributors (e.g., co-authors, reviewers).

Conclusion

Live algorithm services are increasingly developed and used as components in other algorithmic services through the cloud. This in-depth study of a live imagery-analytics service examined the processes of regenerating operations and perpetuating disruptions that kept a service live and growing in conditions of heterogeneous, concurrent, and indefinite updating. Examining the practices of regenerative coordination has generated further understanding of the work increasingly required of members and algorithms (e.g., detecting, preserving, restoring) working on a live algorithmic service, and identified the consequences of working through indefinite updates and disruptions. As continuity and consistency of service become more prominent objectives when coordinating distributed and collective work, examining how and

why members extend the timeframe and objectives of specific updates despite the stresses and uncertainties these perpetuate is critical to our understanding of how regenerative coordination practices can be leveraged to mitigate the overwork and under-recognition workers may experience from them.

References

- Acker, A., & Beaton, B. (2016). Software update unrest: The recent happenings around Tinder and Tesla. *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 1891–1900.
- Adler, P. S. (1995). Interdepartmental Interdependence and Coordination: The Case of the Design/Manufacturing Interface. *Organization Science*, 6(2), 147–167. JSTOR.
- Alaimo, C., & Kallinikos, J. (2020). Managing by Data: Algorithmic Categories and Organizing. *Organization Studies*, 0170840620934062. <https://doi.org/10.1177/0170840620934062>
- Arditi, D. (2018). Digital subscriptions: The unending consumption of music in the digital era. *Popular Music and Society*, 41(3), 302–318.
- Barley, S. R. (1986). *Technology as an Occasion for Structuring; Evidence from Observations of CT Scanners and the Social Order of Radiology Departments*. 32.
- Barley, W. C. (2015). Anticipatory work: How the need to represent knowledge across boundaries shapes work practices within them. *Organization Science*, 26(6), 1612–1628.
- Barley, W. C., Leonardi, P. M., & Bailey, D. E. (2012). Engineering Objects for Collaboration: Strategies of Ambiguity and Clarity at Knowledge Boundaries. *Human Communication Research*, 38(3), 280–308. <https://doi.org/10.1111/j.1468-2958.2012.01430.x>
- Barley, W. C., Treem, J. W., & Leonardi, P. M. (2020). Experts at coordination: Examining the performance, production, and value of process expertise. *Journal of Communication*, 70(1), 60–89.
- Bartholet, R. G., Brogan, D. C., Reynolds Jr, P. F., & Carnahan, J. C. (2004). *In search of the philosopher's stone: Simulation composability versus component-based software design*. VIRGINIA UNIV CHARLOTTESVILLE DEPT OF COMPUTER SCIENCE.
- Beane, M., & Orlikowski, W. J. (2015). What difference does a robot make? The material enactment of distributed coordination. *Organization Science*, 26(6), 1553–1573.
- Bechky, B. A. (2006). Gaffers, Gofers, and Grips: Role-Based Coordination in Temporary Organizations. *Organization Science*, 17(1), 3–21. <https://doi.org/10.1287/orsc.1050.0149>
- Bechky, B. A., & Okhuysen, G. A. (2011). Expecting the unexpected? How SWAT officers and film crews handle surprises. *Academy of Management Journal*, 54(2), 239–261.
- Becker, H. S. (2008). *Tricks of the trade: How to think about your research while you're doing it*. University of Chicago press.
- Bhattacharjee, A. (2001). Understanding Information Systems Continuance: An Expectation-Confirmation Model. *MIS Quarterly*, 25(3), 351–370. <https://doi.org/10.2307/3250921>
- Bolici, F., Howison, J., & Crowston, K. (2016). Stigmergic coordination in FLOSS development teams: Integrating explicit and implicit mechanisms. *Cognitive Systems Research*, 38, 14–22.
- Bolton, D. K., & Friedl, M. A. (2013). Forecasting crop yield using remotely sensed vegetation indices and crop phenology metrics. *Agricultural and Forest Meteorology*, 173, 74–84.
- Brosius, M., Haki, M. K., Aier, S., & Winter, R. (2017). A Literature Review of Coordination Mechanisms: Contrasting Organization Science and Information Systems Perspectives. *Enterprise Engineering Working Conference*, 220–233.
- Bruns, H. C. (2013). Working alone together: Coordination in collaboration across domains of expertise. *Academy of Management Journal*, 56(1), 62–83.

- Busco, C., & Quattrone, P. (2018). In Search of the “Perfect One”: How accounting as a maieutic machine sustains inventions through generative ‘in-tensions.’ *Management Accounting Research*, 39, 1–16.
- Cetina, K. K., & Bruegger, U. (2002). Traders’ engagement with markets. *Theory, Culture & Society*, 19(5–6), 161–185.
- Ching, K., Forti, E., & Rawley, E. (2019). Extemporaneous Coordination in Specialist Teams: The Familiarity Complementarity. *SSRN Electronic Journal*.
<https://doi.org/10.2139/ssrn.3491456>
- Christianson, M. K. (2019). More and Less Effective Updating: The Role of Trajectory Management in Making Sense Again. *Administrative Science Quarterly*, 64(1), 45–86.
<https://doi.org/10.1177/0001839217750856>
- Christin, A. (2017a). Algorithms in practice: Comparing web journalism and criminal justice. *Big Data & Society*, 4(2), 2053951717718855.
- Christin, A. (2017b). Algorithms in practice: Comparing web journalism and criminal justice. *Big Data & Society*, 4(2), 2053951717718855. <https://doi.org/10.1177/2053951717718855>
- Christin, A. (2018). Counting clicks: Quantification and variation in web journalism in the United States and France. *American Journal of Sociology*, 123(5), 1382–1415.
- Christin, A. (2020). The ethnographer and the algorithm: Beyond the black box. *Theory and Society*. <https://doi.org/10.1007/s11186-020-09411-3>
- Curchod, C., Patriotta, G., Cohen, L., & Neysen, N. (2019). Working for an algorithm: Power asymmetries and agency in online work settings. *Administrative Science Quarterly*, 0001839219867024.
- Cusumano, M. A. (2008). The changing software business: Moving from products to services. *Computer*, 41(1), 20–27.
- Dahlander, L., & O’Mahony, S. (2011). Progressing to the Center: Coordinating Project Work. *Organization Science*, 22(4), 961–979. <https://doi.org/10.1287/orsc.1100.0571>
- DiBenigno, J., & Kellogg, K. C. (2014). Beyond occupational differences: The importance of cross-cutting demographics and dyadic toolkits for collaboration in a US hospital. *Administrative Science Quarterly*, 59(3), 375–408.
- Dourish, P. (2016). Algorithms and their others: Algorithmic culture in context. *Big Data & Society*, 3(2), 2053951716665128.
- Du, J., Belderbos, R., & Leten, B. (2020). Temporal Dynamics in Multipartner Coordination: Sequentiality and Simultaneity in R&D Projects. *Academy of Management Proceedings*, 2020(1), 22024.
- Faraj, S., & Sproull, L. (2000). Coordinating expertise in software development teams. *Management Science*, 46(12), 1554–1568.
- Faraj, S., & Xiao, Y. (2006). Coordination in fast-response organizations. *Management Science*, 52(8), 1155–1169.
- Fayard, A.-L., & Metiu, A. (2014). The role of writing in distributed collaboration. *Organization Science*, 25(5), 1391–1413.
- Fayol, H., & STORRS, C. (1949). *Administration Industrielle Et Générale. General and Industrial Management... Translated... By Constance Storrs, Etc.* Sir Isaac Pitman & Sons.
- Feldman, M. S., & Orlikowski, W. J. (2011). Theorizing Practice and Practicing Theory. *Organization Science*, 22(5), 1240–1253. <https://doi.org/10.1287/orsc.1100.0612>

- Feldman, M. S., & Rafaeli, A. (2002). Organizational routines as sources of connections and understandings. *Journal of Management Studies*, 39(3), 309–331.
- Fine, G. A., Morrill, C., & Surianarain, S. (2009). *Ethnography in organizational settings*.
- Garud, R., Jain, S., & Tuertscher, P. (2008). Incomplete by Design and Designing for Incompleteness. *Organization Studies*, 29(3), 351–371.
<https://doi.org/10.1177/0170840607088018>
- Geiger, D., Danner-Schröder, A., & Kremser, W. (2020). Getting Ahead of Time—Performing Temporal Boundaries to Coordinate Routines under Temporal Uncertainty. *Administrative Science Quarterly*, 0001839220941010.
<https://doi.org/10.1177/0001839220941010>
- Geiger, R. S. (2014). Bots, bespoke, code and the materiality of software platforms. *Information, Communication & Society*, 17(3), 342–356.
<https://doi.org/10.1080/1369118X.2013.873069>
- Glaser, V. L., Pollock, N., & D’Adderio, L. (2021). The Biography of an Algorithm: Performing algorithmic technologies in organizations. *Organization Theory*, 2(2), 26317877211004610. <https://doi.org/10.1177/26317877211004609>
- Gray, M. L., & Suri, S. (2019). *Ghost work: How to stop Silicon Valley from building a new global underclass*. Eamon Dolan Books.
- Harrison, S. H., & Rouse, E. D. (2014). Let’s dance! Elastic coordination in creative group work: A qualitative study of modern dancers. *Academy of Management Journal*, 57(5), 1256–1283.
- Hinds, P. J., & Bailey, D. E. (2003). Out of sight, out of sync: Understanding conflict in distributed teams. *Organization Science*, 14(6), 615–632.
- Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Pearson Education.
- Jarzabkowski, P. A., Lê, J. K., & Feldman, M. S. (2012). Toward a theory of coordinating: Creating coordinating mechanisms in practice. *Organization Science*, 23(4), 907–927.
- Jordan, S., & Messner, M. (2012). Enabling control and the problem of incomplete performance indicators. *Accounting, Organizations and Society*, 37(8), 544–564.
- Kaplan, S., & Orlikowski, W. J. (2013). Temporal Work in Strategy Making. *Organization Science*, 24(4), 965–995. <https://doi.org/10.1287/orsc.1120.0792>
- Karunakaran, A. (2021). *In Cloud We Trust? Coopting Occupational Gatekeepers to Produce Normalized Trust in Platform-mediated Interorganizational Relationships*.
- Kellogg, K. C., Orlikowski, W. J., & Yates, J. (2006). Life in the trading zone: Structuring coordination across boundaries in postbureaucratic organizations. *Organization Science*, 17(1), 22–44.
- Kitchin, R. (2017). Thinking critically about and researching algorithms. *Information, Communication & Society*, 20(1), 14–29.
- Klein, K. J., Ziegert, J. C., Knight, A. P., & Xiao, Y. (2006). Dynamic Delegation: Shared, Hierarchical, and Deindividualized Leadership in Extreme Action Teams. *Administrative Science Quarterly*, 51(4), 590–621. <https://doi.org/10.2189/asqu.51.4.590>
- Kremser, W., & Blagojev, B. (2020). The Dynamics of Prioritizing: How Actors Temporally Pattern Complex Role–Routine Ecologies. *Administrative Science Quarterly*, 0001839220948483. <https://doi.org/10.1177/0001839220948483>

- Kwan, M.-P. (2016). Algorithmic geographies: Big data, algorithmic uncertainty, and the production of geographic knowledge. *Annals of the American Association of Geographers*, 106(2), 274–282.
- Langley, A. N. N., Smallman, C., Tsoukas, H., & Van de Ven, A. H. (2013). Process studies of change in organization and management: Unveiling temporality, activity, and flow. *Academy of Management Journal*, 56(1), 1–13.
- LeBaron, C., Christianson, M. K., Garrett, L., & Ilan, R. (2016). Coordinating flexible performance during everyday work: An ethnomethodological study of handoff routines. *Organization Science*, 27(3), 514–534.
- Lebovitz, S., Levina, N., & Lifshitz-Assaf, H. (2021). Is AI ground truth really “true”? The dangers of training and evaluating AI tools based on experts’ know-what. *Management Information Systems Quarterly*.
- Lifshitz-Assaf, H., Lebovitz, S., & Zalmanson, L. (2020). Minimal and Adaptive Coordination: How Hackathons’ Projects Accelerate Innovation without Killing it. *Academy of Management Journal*, ja.
- Lusch, R. F., & Nambisan, S. (2015). Service innovation. *MIS Quarterly*, 39(1), 155–176.
- Majchrzak, A., Jarvenpaa, S. L., & Hollingshead, A. B. (2007). Coordinating expertise among emergent groups responding to disasters. *Organization Science*, 18(1), 147–161.
- Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1), 87–119. <https://doi.org/10.1145/174666.174668>
- Manikas, K. (2016). Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117, 84–103.
- Mengis, J., Nicolini, D., & Swan, J. (2018). Integrating knowledge in the face of epistemic uncertainty: Dialogically drawing distinctions. *Management Learning*, 49(5), 595–612.
- Morreale, F., & Eriksson, M. (2020). “My Library Has Just Been Obliterated”: Producing New Norms of Use Via Software Update. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–13. <https://doi.org/10.1145/3313831.3376308>
- Nagaraj, A., & Stern, S. (2020). The Economics of Maps. *Journal of Economic Perspectives*, 34(1), 196–221.
- Neff, G., & Stark, D. (2004). *Permanently beta: Responsive organization in the Internet era*.
- Nieborg, D. B., & Poell, T. (2018). The platformization of cultural production: Theorizing the contingent cultural commodity. *New Media & Society*, 20(11), 4275–4292.
- Okhuysen, G. A., & Bechky, B. A. (2009). 10 coordination in organizations: An integrative perspective. *Academy of Management Annals*, 3(1), 463–502.
- Orlikowski, W. J., & Scott, S. V. (2014). What happens when evaluation goes online? Exploring apparatuses of valuation in the travel sector. *Organization Science*, 25(3), 868–891.
- Orlikowski, W. J., & Scott, S. V. (2015). The Algorithm and the Crowd. *MIS Quarterly*, 39(1), 201–216.
- Pachidi, S., Berends, H., Faraj, S., & Huysman, M. (2020). Make Way for the Algorithms: Symbolic Actions and Change in a Regime of Knowing. *Organization Science*.
- Parnas, D. L., Clements, P. C., & Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 3, 259–266.
- Pine, K. H., & Mazmanian, M. (2017). Artful and contorted coordinating: The ramifications of imposing formal logics of task jurisdiction on situated practice. *Academy of Management Journal*, 60(2), 720–742.

- Rahman, H. A. (2021). The Invisible Cage: Workers' Reactivity to Opaque Algorithmic Evaluations. *Administrative Science Quarterly*, 00018392211010118.
- Sachs, S. E. (2019). The algorithm at work? Explanation and repair in the enactment of similarity in art data. *Information, Communication & Society*, 1–17.
- Salovaara, A., Lyytinen, K., & Penttinen, E. (2019). High reliability in digital organizing: Mindlessness, the frame problem, and digital operations. *MIS Quarterly*.
- Sanchez, R., & Mahoney, J. T. (1996). Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal*, 17(S2), 63–76.
- Savitsky, K., Medvec, V. H., & Gilovich, T. (1997). Remembering and regretting: The Zeigarnik effect and the cognitive availability of regrettable actions and inactions. *Personality and Social Psychology Bulletin*, 23(3), 248–257.
- Schakel, J.-K., van Fenema, P. C., & Faraj, S. (2016). Shots Fired! Switching Between Practices in Police Work. *Organization Science*, 27(2), 391–410.
<https://doi.org/10.1287/orsc.2016.1048>
- Seaver, N. (2017). Algorithms as culture: Some tactics for the ethnography of algorithmic systems. *Big Data & Society*, 4(2), 205395171773810.
<https://doi.org/10.1177/2053951717738104>
- Seaver, N., Vertesi, J., & Ribes, D. (2019). Knowing algorithms. In *DigitalSTS* (pp. 412–422). Princeton University Press.
- Sergeeva, A. V., Faraj, S., & Huysman, M. (2020). Losing Touch: An Embodiment Perspective on Coordination in Robotic Surgery. *Organization Science*.
- Shestakofsky, B. (2017). Working algorithms: Software automation and the future of work. *Work and Occupations*, 44(4), 376–423.
- Spradley, J. P. (2016). *Participant observation*. Waveland Press.
- Star, S. L., & Strauss, A. (1999). Layers of silence, arenas of voice: The ecology of visible and invisible work. *Computer Supported Cooperative Work (CSCW)*, 8(1–2), 9–30.
- Stark, D., & Pais, I. (2020). Algorithmic management in the platform economy. *Sociologica*, 14(3), 47–72.
- Stephens, J. P. (2020). How the Show Goes On: Using the Aesthetic Experience of Collective Performance to Adapt while Coordinating. *Administrative Science Quarterly*, 0001839220911056.
- Švelch, J. (2019). Resisting the perpetual update: Struggles against protocological power in video games. *New Media & Society*, 21(7), 1594–1612.
- Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software: Beyond object-oriented programming*. Pearson Education.
- Terwiesch, C., Loch, C. H., & Meyer, A. D. (2002). Exchanging Preliminary Information in Concurrent Engineering: Alternative Coordination Strategies. *ORGANIZATION SCIENCE*, 13(4), 19.
- Tuertscher, P., Garud, R., & Kumaraswamy, A. (2014). Justification and Interlaced Knowledge at ATLAS, CERN. *Organization Science*, 25(6), 1579–1608.
<https://doi.org/10.1287/orsc.2013.0894>
- Valentine, M. A., & Edmondson, A. C. (2015). Team scaffolds: How mesolevel structures enable role-based coordination in temporary groups. *Organization Science*, 26(2), 405–422.
- Vis, F. (2013). A critical reflection on Big Data: Considering APIs, researchers and tools as data makers. *First Monday*.

Weiss, R. S. (1995). *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster.

Whyte, J., Lindkvist, C., & Jaradat, S. (2016). Passing the baton? Handing over digital data from the project to operations. *Engineering Project Organization Journal*, 6(1), 2–14.
<https://doi.org/10.1080/21573727.2015.1115396>