Written Communication
1–49
© 2020 SAGE Publications
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/0741088320968061
journals.sagepub.com/home/wcx
SAGE

Writing With Data: A Study of Coding on a Data-Journalism Team

Chris Aaron Lindgren[1]

Abstract

Coding has typically been understood as an engineering practice, where the meaning of code has discrete boundaries as a technology that does precisely what it says. Multidisciplinary code studies reframed this technological perspective by positing code as the latest form of writing, where code's meaning is always partial and dependent on situational factors. Building out from this premise, this article theorizes coding as a form of *writing with data* through a qualitative case study of a web developer's coding on a data-journalism team. I specifically theorize code as a form of intermediary writing to examine how his coding to process and analyze data sets involved the construction and negotiation of emergent problems throughout his coding tasks. Findings suggest how he integrated previous coding experience with an emerging sense of how code helped him write and revise the data. I conclude by considering the implications of these findings and discuss how writing and code studies could develop mutually informative approaches to coding as a situated and relational writing activity.

Keywords

code studies, software studies, computational literacy, materiality, intermediation, case study, data processing

[1]Virginia Polytechnic Institute and State University, Blacksburg, USA

Corresponding Author:
Chris Aaron Lindgren, Department of English, Virginia Polytechnic Institute and State University, 207 Shanks Hall, 181 Turner Street NW, Blacksburg, VA 24061, USA.
Email: lindgren@vt.edu

So my title is "Frontend Developer." Title-wise, it's geared towards programming. What that means is that I end up building interfaces—essentially all web-based interfaces. I'd say that's about 50% of the job. The other 50% is data-related: data processing, analysis, and just getting data, which requires more effort than one would think. And most of the time on these activities, I'm interfacing with our data team or reporters in the newsroom.

—Ray (web developer)[1]

Bridging Competing Perspectives on Code

Meet Ray: a web developer who works with reporters and editors on a data-journalism team for a large news organization.[2] This team creates data-driven news stories by integrating large data sets into journalistic inquiries and narrative reporting, publishing stories akin to ProPublica and FiveThirtyEight. In the epigraph above, drawn from an ethnographic study of Ray's coding activity, he described how much of his role involved "building interfaces," since his code yielded web-bound data visualizations called *interactives*. While Ray has written code in a variety of programming languages, he wrote with JavaScript (JS) in this context to help reporters more quickly and accurately create these interactives.[3] He also used a "building" analogy to describe the "other 50%" of his job: coding with data sets. Ray never directly described his coding with data as a text-based reading or writing activity. For instance, in an observational interview, instead of "reading data," he described how data must be "looked at" for multiple reasons. And, instead of "writing code that helps him write data," he described data processing as a "reverse-engineering of the data." Even so, while Ray's personal descriptions of his coding drew on building and engineering analogies, he did not presuppose that code-as-technology automated away all the meaning making involved in his processing and analysis work. As he remarked, it "requires more effort than one would think."

Throughout the course of the study, I witnessed how Ray's code-as-technology perspective seemed to push another dimension of his coding into the background. Specifically, his coding was supported by large textual data sets that his code helped him to write differently. For example, coding enabled Ray to automate the writing of information from files, so he could combine selected aspects of one set with another. He noted how programming languages, such as JS, automated certain ways of "reading in" structured data. Ray used this type of reading to describe how JS automated the parsing of standardized file formats into JS-specific structured data (see Appendix A). From this mundane parsing of data files into more malleable JS-structured content, he wrote code that reduced and transformed data sets from their original form, so he could conduct analyses important to the changing goals of the team. By analyzing Ray's coding as a form of writing jointly with data, I observed how his coding wrote structured information from varying sources and origins, which involved dynamic meaning-making strategies to write new data sets that contextualized information for a project. In this article, I approach coding as an intermediary form of writing with data, wherein coders negotiate a tension between coding as a technical process and coding as situated and relational writing activity.

Complicating Technological Perspectives of Code

Much like Ray's technological description of his role on the data-journalism team, code and coding are often defined and understood in terms of what they build—its outputs: digital tools, interfaces, and technologies. Across much of the existing research on coding, the theoretical and pragmatic aims of engineering, mathematics, and the sciences dominate its insights. For instance, Vee (2017) suggests that "[c]omputer science values theoretical principles of design and abstraction, and software engineering emphasizes modularity, reusability, and clarity in code" (p. 15). Researchers and practitioners from computer science and engineering (Brown, 2006; Knuth, 1968; Ko, 2016; Ko et al., 2015; Mei, 2014; Winograd & Flores, 1986) and the social sciences (Higgins, 2007; Prior et al., 2006) have expressed similar perspectives. Existing empirical studies often invest in broader generalizable effects of software development activities to support coding as the engineering of automation and development of end-user technologies, more efficient and secure algorithmic protocols, or more efficient debugging strategies of large software systems. These generalizing aims are important for their respective goals and concepts that demarcate computer science and engineering's building of novel automated computational systems in the most efficient way. Yet, if these engineering aims are the stronghold of theories of code and data, then it should be no surprise how they are often understood in terms of engineered automation alone.

Prevailing technological perspectives offer important insights into code, since code *is* an object of engineered automation. And yet, this perspective has not fully captured the complexity of coding as a situated activity. For instance, computer science and human-computer interaction researchers (Ko et al., 2007; Sillito et al., 2008) have investigated what questions software developers ask as they code, but their findings indicate a tight focus on the technical state of the data as it manifests across a codebase. The broadest definition of contextual questions included examples, such as "Which [data] type represents this domain concept or this UI [(user-interface)] element or action?" (Sillito et al., 2008, p. 439). Context is bound by terms, such as "domain concept," that maintains relations between coders and code to the specific technological feature that the code produces during a particular task. In this article, I illustrate how Ray's coding involved making sense of computational actions (e.g., how code automates transformative actions on data) with similar questions across an ensemble of tools, written artifacts, and other people. Yet I also show how his coding involved his sense of contextual and situated concerns of the data-journalism team—factors that are normally deemed outside the scope of the code-as-technology perspective. I argue that accounting for these features complicates narrow approaches to code and data and usefully expands what counts as writing and literate practice in a technical workplace environment.

This case study draws on multidisciplinary scholarship from across software studies and literacy studies, which have argued that computer code and data are forms of writing that exceed the technological perspective of code. Specifically, I begin by reviewing scholarship that characterizes coding as a form of writing. In essence, these scholars theorize the meaning of code and data as situationally and relationally produced by the people coding with data, as opposed to code and data having discrete technological meaning bound to computational action alone. This scholarship I review usefully approaches code as an intermediating textual resource that facilitates meaning making during every situation, but it is yet to provide substantial evidence about how meaning making happens in situ or how we might go about studying coding empirically. Drawing on this body of work, I then develop a framework that I use to isolate aspects of Ray's meaning-making strategies and theorize them as a form of writing with code and data across multiple coding technologies.

Code and Data: Textual Resources for Sense Making

Computer code has often been positioned as the first form of writing that *does what it says* with great precision. Kittler (1997, 1999) and later Manovich (2002) and Galloway (2006) argued that computer code achieves a Saussurian fixed-code system, since it reduces signification to a binary system of tightly defined voltage difference: A signifier is either something or nothing. Kittler made rather bombastic claims about computational media as autonomous and deterministic of any situation, where writing (and other media) have been dissolved by these binary operations. Numerous scholars have elaborated on the transgressions of such claims (Chun, 2005; Vee, 2017; Winograd & Flores, 1986).

Hayles (2005) conceded that computer code can be reduced to digital binary signifiers, because a computer's architecture requires such rigid data encoding/decoding to function. Yet this is where her agreement ends. She explains the negative consequences of these reductive claims and classifies them as the *regime of computation*, which includes the broader acceptance that computation, in the form of computer code, "acquires special, indeed universal, significance" (p. 27). Her rebuttal to the technological perspective explains how it conflates the machinic encoding/decoding of information into a broader theory of unambiguity, wherein Kittler's theory of computational logic attempts to cultivate a deterministic situation and ignore the persistent integration of communicative labor. Machine code may be figured through layered binaries, but its meaning and production are not linear processes that begin and end as a binary.

Hayles rebuilds the bridge between people, meaning making, and code. She explained how these factors are always intermediating, that is, interacting, to layer these binaries on new layers of representational units that a person's code expresses and performs. She described how the complexity of computational systems often become synonymous with thinking, stating that

> [coded] components can be structured so as to build up increasing levels of complexity, eventually arriving at complexity so deep, multilayered, and extensive as to simulate the most complex phenomena on earth, from turbulent flow and multiagent social systems to reasoning processes one might legitimately call thinking. (p. 17)

Hayles contends that this multilayering of code operations and stored digital data underlies the myth that code is "the discourse system that mirrors what happens in nature and that generates nature itself" (p. 27). She positions "intermediation" (p. 31) as a theory to explain how people, code, writing, speech, and other forms of media interact with each other in complex ways that have yet to be studied in more detail.

Other examples of this regime include Manovich's (2002) claim that computational media is composed of two discrete layers: cultural and computational. His distinction relies on his concept of transcoding, which he defines as the computational act to transform data from one format to another. For instance, word-processing software such as Microsoft Word transcodes typewriters, the printed page, and its accompanying practices. He argued that transcoding "the computer layer will affect the cultural layer" (p. 46). This theory of software has helped scholars examine the relationships between older and newer media. But, Manovich's clean separation between culture and the computational operations of code and data uphold problematic assumptions about how and where meaning is produced and sustained.

Indeed, Manovich asserts that knowing how to code is synonymous with examining and understanding the entirety of these programmable layers. Hayles (2005) and Chun (2005) both disagree that an examination of computer code will make visible the complete nature of code and data. Specifically, Chun isolates the central issue with Manovich's theorizing of software and culture as distinct layers, stating that Manovich "focuses on static data and treats computation as a mere translation" (2005, p. 46) of said data. Code and data should never be considered static, and code's computational acts are deceptively not so discretely autonomous and transparently mapped for all to know in its entirety. Nothing can be datafied or modeled perfectly, and consequently transcoding must also be considered as intermediated by dynamic social-technological systems.[4]

Writing has often been positioned against this regime of computation that sustains myths about the transparent meaning of code and data. Chun (2011) links persistent myths about code and data, as being self-contained truths, with similar myths about writing as generalizable across contexts. She admits that computer code has automated numerous forms of knowledge labor, such as the storage, retrieval, and other custom dexterities associated with data. By dexterities with data, recall how writing code is linked to automating the writing of structured data with computers. Coding has historical connections with transforming textual, tabulated punched-card data into digital memory and back into printed reports. While this history extends beyond the scope of this article, Admiral Grace Hopper (1978), who is considered one of the initial people to develop programming language interpreters, once remarked how the coders of her time were attempting to encode their physical manual dexterities with punched-card code that wrote code for them based on their recurrent situations processing and analyzing data.[5] This automated form of writing helped coders more quickly compute and write data at greater speed and scale. Despite this automation, code and data have never and will never transcend into a Platonic, autonomous model of information, because "coding still means producing a mark, a writing, open to alteration/iteration rather than an airtight anchor" (Chun, 2011, p. 25). Material and symbolic action is still required, which opens up questions about how code is not *the source* of information, but, as Chun argued, it is "more accurately a re-source" (p. 25) that integrates external situational factors.[6]

These studies offer the tenets to a new research agenda. Specifically, akin to any utterance, the material presence of code and data does not presume a determined representation or interpretation—only a durable carrier and transformation of code and data. This durability of code and data often produces a veneer of more austere textual modes of production and use than traditional notions of writing. But, just as any other form of writing, the texts that coders and computers relationally read and write can never capture or contain the entire account of their production, intent, or uptake. Consequently, the next goal to understand where the meaning of code and data reside involves studying how coders negotiate the joint textual development between code and data in relationship to their context. Hayles (2005) theorized the concept of intermediation to call others to study how code interacts with people and "legacy" (p. 38) forms of language, such as traditional notions of what counts as writing. However, media and software studies scholars have yet to provide a clearer path for applying it. This intermediation across people, technologies, and media involves a negotiated process of making meaning, where researchers can account for a wider range of strategies that writers do not often inscribe in their texts for others to engage with after-the-fact. To provide such a path, I draw on scholarship from writing studies to construct a theoretical framework that positions coding as an intermediary form of writing with data.

Ensembled Text Senses and Distributions: A Framework for Studying Coding as Writing

To examine the materiality of writing code means understanding that computer code and data cannot be reduced to their linguistic signs, and it means understanding that code, even as a digital data referent, does not exist as a 1:1 relationship between machine and code. If this 1:1 relationship existed, code would always be consciously understood by the person writing it, always be self-evident to other audiences, always be static and never in need of revision, and thereby always be independent of historical and situational matters. If such plainness were possible, all code and its structured data would be premeditated, unchanging, deliberate, and predictable. Rather than conceptualizing code in this way, I instead approach it as an intermediary form of writing with data that confronts the process of negotiating meaning across writing activities. I develop this framework with three concepts, which share a methodological lineage: text sense (Haas, 1996), work ensembles (Bracewell & Witte, 2003), and spectrum of durability (Clayson, 2018).

Haas (1996) developed the concept of *text sense* to describe the tacit internalizations of the writer's material goal: How writers sensitize themselves to socially recognizable versions of their texts, such as a business report. Text sense churned up out of writers having difficulty in adjusting to writing with novel personal computers of the time (1980s). Specifically, the computer's word-processing interface interrupted the relationship between writers and their habituated reliance on print-based materials that supported their meaning making. Consequently, the writers developed a strategy to print out their *texts so far*, so they could better understand the text's development. A focus on text sense helps to show how writers naturalize their unfolding sense of text in relation to their efforts to materialize specific goals in the context of their writing activity.

Empirically, *senses* are inferred by researchers from the writer's situated act to materialize language. Smagorinsky (2001) traced the etymology of *sense* from Vygotsky's (1987) studies of concept formation in children in order to explain this inferential relationship. Vygotsky, as other scholars have explained in more detail (Imbrenda, 2016; White, 2014; Witte, 1992), crafted a Marxist dialectical theory of mediation to better understand how thought develops in coordination with our social and material conditions. Smagorinsky explains that Vygotksy posited *smysl* (sense) as that which is "yet unarticulated, being instead the storm cloud of thought that produces the shower of words" (p. 145). Sense accompanies what Vygotsky called *znachenie*, which Smagorinsky translates as *articulation*: "the zone of meaning available in represented form, corresponding to the notion of a sign, regardless of modality" (p. 145). The previous English translation of *znachenie* was meaning, but Smagorinsky wanted to convey its relational aspects between people and texts. By tethering *sense* and *articulation*, Smagorinsky (2001) argued that "readers and texts share a cultural cognizance" (p. 146) that does not predetermine a meaning but supports the potential for shared-enough understandings. The connection between text sense and articulation guides studies of writing to focus on how people develop conceptual links between their text sense and the materialized forms of writing that they produce.

Smagorinsky translated Vygotsky's *znachenie* as "articulation," but other writing researchers have developed this concept under a different set of terms. For instance, instead of using articulation to describe a writer's external representations, Bracewell and Witte (2003) extended Vygotsky's *zone of meaning* to develop a methodological construct of *work ensembles*. This unit of analysis pinpoints how writers in professional settings articulate their writing goals as tasks across ensembles of people, artifacts, and tools. Among other theoretical and empirical concerns with activity theory,[7] they argued that ensembles help researchers examine the dynamic strategies that writers use, when articulating goals into tasks. As other studies have found (Bazerman et al., 2017; Bizzell, 2003; Byrd, 2019; Opel & Hart-Davidson, 2019; Pigg, 2014), writers encounter unpredictable problems when carrying out explicit goals, and they adapt to these emergent problems with support from dynamic ensembles. Bizzell (2003) notes this recurrent writing problem, where the lines blur between text planning and production when writers define their problems as they interact with their social-material environments (p. 403).

Bracewell and Witte's focus on tasks and ensembles, and the semiotic relation between them, is also informed by Hutchins's (1995) theory of distributed cognition. Hutchins uses the concept of distributed cognition to highlight how meaning making is connected to social-material environments: contextual ensembles of accompanying people, objects, and technologies (p. 27). Rather than using articulation as the term to describe how writers materialize their textual activity across a range of material modes, Bracewell and Witte gesture toward the use of *distribution* as the central mediational act across the ensemble. In other words, writers and their ensembles foster intermediary relationships between thinking and writing activity to distribute their text sense "across a series of representational media" (Hutchins, 1995, as cited in Bracewell & Witte, 2003, p. 529). What is pertinent for writing studies is how ensembles guide researchers to examine this relationship between sense and distribution. Writing goals will be performed as tasks with an accompanying ensemble, which can be used to examine how writers negotiate the meaning and production of texts when encountering and developing emergent problems.[8]

Just as Bracewell and Witte developed the concept of ensemble to study the intermediated relationship between senses and distributions, Clayson (2018) developed the *spectrum of durability* to explore the wide range of material and embodied distributions that writers produce as they develop their text sense. Clayson foregrounded the difficulties that writers faced when "transforming multiple representations into prose" (p. 175) and took up Hutchins's (1995) theory of distributed cognition to study the embodied gestures of a team collaboratively planning a report document. Clayson illustrated how people negotiate emergent problems during a writing session, acting, in Hutchins's (1995) words, as "malleable and adaptable coordinating tissue" (p. 219) in tandem with their durable ensemble of technological media.[9] In Clayson's case, this spectrum ranged from ephemeral speech and accompanying hand gestures, to more durable lists and outlines produced on a whiteboard, to the eventual drafting of a report genre form for intended audiences. Across the creation and uses of report-oriented distributed media, she found that writers shored up a sense of the "'the text produced so far'" (Flower & Hayes quoted in Clayson, 2018, p. 158) by externalizing the report's shape and content. Clayson's spectrum usefully highlights the often-neglected textual representations that importantly help writers develop and maintain "targeted senses of the texts" (p. 178).

The framework I am developing here brings together three principal concepts that have emerged from studies of writing as a situated activity: text sense, work ensembles, and spectrum of durability. I draw on these concepts to explore and theorize what Ray and the team referred to as coding to *slice* data,[10] which Ray performed during data-processing and analysis tasks. In general terms, slicing involves reducing a data set and transforming it in multiple ways, whether aggregating its index, computing new variables, or some combination of both. By studying Ray's slicing activity as a form of intermediary writing, I show how coding is a constructive means whereby he negotiated emergent problems across an ensemble that produced a range of textual distributions. Specifically, I examine the following questions:

1. What textual representations does Ray produce during his coding to slice data?
2. How does Ray's coding negotiate emergent problems by distributing texts across an ensemble of people, tools, and artifacts?
3. What senses of the code and data as texts can be inferred from his meaning making work with his coding ensemble?

By studying Ray's *text senses* and range of *distributions* across his coding *ensemble*, I describe how he used coding to negotiate emergent problems. Overall, this case study demonstrates that code does not inscribe all of the factors that intermediate the written production of code and data.

Case Background: Coding on a Data Team

The present study is drawn from a larger case study of Ray's coding on a data-journalism team. In this article, I examine a subset of this broader case by focusing on a particular form of coding, *slicing* data sets, to better understand how he negotiated emergent problems throughout the task. Ray's slicing transformed data across a spectrum of textual distributions in the service of the news story.

Developers like Ray, whose previous coding history involved database administration and web development, are now finding careers within an editorial agenda that uses, produces, and contextualizes large swathes of data for news narrative ends. Data sets have become easier to access, process, analyze, visualize, and create, thanks in large part to computer coding and the internet. The data processing that I examine in this article is often characterized as the mundane, yet unavoidable, part of the data-driven process. Slicing requires insight into how to write code that transforms the data structure and content into a desired new state for analysis, which can prove difficult sometimes. According to a survey of over 16,000 data-science practitioners (Kaggle, 2017), Ray is not alone. People who code with data sets report that "dirty data" is the most common barrier that they face at work before their desired analysis.

Ray telecommuted and worked from his home during my observations. He stayed in close contact with his team through stand-up morning meetings over Google Hangout video chats and the Slack messaging application throughout the work day. He is considered one of two developers on this team, which also includes an editor, producer, and multiple reporters. However, the reporters are not officially on the data-journalism team. Instead, reporters work within the broader newsroom and come to the team for data-driven support. This study focuses on Ray's work with the team's editor, Vince, and producer, Jun, who worked with Ray in the scope of the reduced data for this article. Their names have been changed to maintain confidentiality.

Ray's coding supported the narrative work of reporters and editors on the data team and other colleagues across the entire news organization. Many of the results reported in this article concern Ray's interactions with Jun and Vince. In a semistructured interview, Ray noted how Jun's producer role meant that "she does all types of stuff, which includes some code-related work [with the data]." He also noted how much of his situations with Jun involved her doing some initial analysis and then coming to Ray at an insurmountable boundary with data or coding. Regarding Vince, Ray stated how "[Vince] does some coding stuff, too, but his role is mostly just making sure things are good and accurate."

During my observational period, Ray worked on roughly 11 projects.[11] Four of the 11 total projects were specifically developed by Ray's team, three projects supported reporters from the broader newsroom, and the remaining four projects supported other organizational services (see Table 1). He coded across numerous types of projects, each with their own objectives and exigencies.

The coding problems Ray engaged involved some overlapping situational features. Sometimes a reporter had been working on some data, but they would ask him to verify the data and visualize it for the news report. Other times, reporters had only a germ of a story and multiple large sets of data. In such a situation, Ray's coding helped reporters develop a data-driven angle to their news narrative. I also observed Ray code to collect or import data into databases that archived troves of data sets on dedicated networked servers from various sources. This case study examines Ray's coding to slice data sets, which contextualize the data for the team's publishing goals. In the next section, I describe my method to collect and analyze data about his slicing activity.

Method

In this section, I rationalize the use of a case-study approach, describe the qualitative method used, and the reduced data analyzed in this article. Finally,

Table 1. Summary of Each Project Observed, Its Broader Categories of Ray's Main Objective, and his Goals During the Process.

Project	Ray's main objective	Ray's coding goals
State toxic sites	Develop story about state's toxic sites management	Process, analyze, visualize data
City restoration	Map reporter's data about city's rebuilding program	Code an interactive map with the provided data set
Mapping tool	Develop embeddable map tool for reporters	Ray developed this tool
City payroll	Develop story from newly released data	Process, analyze, visualize data
Transit headway times	Fix archiving script of local transit times	Read existing database script to isolate and resolve issue
Recidivism archive	Process and archive FOIA-requested state data on recidivism	Processed and archived this data
State campaign finances	Reporter wanted database of recurrent data	Ray coded a database that processes and archives this data
Health texting campaign app	Develop a web app that sends, receives, and stores SMS texts for an external journalism team's week-long audience activity	Ray developed, tested, and deployed the web application
Health texting campaign data analysis	After the health texting, Ray needed to find interesting findings for the journalism team to report	Ray processed and analyzed the audience data
Natural disaster effect on transit times	Consult reporter about city government transit data and data team's transit headway data	None. Only discussed data over the phone with the reporter.
Bingo game	Refactor (revise) old code for a web-browser-based Bingo card game for an upcoming celebrity event	Ray read through original HTML, CSS, and JavaScript code. After reading it, he revised all these files for current web standards and readability.
Data importing tool	Develop a generalizable importing tool for large sets of data	Based on Ray's work to archive the political campaign data, he began coding a tool for similar archiving situations

Note. FOIA = Freedom of Information Act, which established a mediating ground for government organizations to share information with the public. SMS = short message service.

I review the more specific analytical methods that I used to develop findings reported herein.

Case Rationale

This case falls under the category of what Yin (2014) refers to as *unusual* (p. 51). Yin argues that a single case can be rationalized, if it confronts the "theoretical norms" (p. 52) about a phenomenon. This case study diverges from normative writing-related activities and objects of study to confront tacit knowledge claims about what constitutes writing. This case also adheres to Dyson and Genishi's (2005) call to illuminate "what some phenomenon means as it is socially enacted within a particular case" (p. 10). For these reasons, I took up a case-study methodology to develop new understandings about computer coding as an intermediating practice to write programmatically with data.

A Modified Grounded-Theory Method: From Expansive to Contractive Data Collection

Table 2 summarizes the method used to collect broader contextual data and finer-grained task-based data for the case study. I used a recursive and iterative process of data collection and analysis over the observational period. These observations incorporated what Farkas and Haas (2012) refer to as a two-movement modified grounded-theory approach: expansive to contractive (pp. 86-89).

My expansive movement collected contextual data, including the following: screen-recordings and field-notes focusing on his situated activity, semistructured interviews about Ray's coding history and "grand tour" (Spradley, 1979) questions about his professional context, and communicative artifacts among the team. After about a month of observations, I noticed Ray's recurrent coding activity to slice data sets and began to sample more data with regard to these activities. For example, my initial semistructured interview included questions about his everyday workplace, his team, and his role on it more broadly, which provided some follow-up questions about his sense of the overall work procedure on the team. This initial expansive effort to valorize the data team's emic goals and Ray's recurrent tasks to slice data sets was followed by the contractive phase to theorize Ray's situated slicing activity as a form of writing.

During the second movement, I analyzed the task-based data collected via think-aloud protocols (TAPs) and retrospective accounts, so I could begin integrating substantive findings about Ray's situated slicing activities with

Table 2. List of Complete Method and Collected Data.

Method	Complete set	Use
Contextual/expansive		
Screen-recordings (audio/video; Mackiewicz & Thompson, 2014; Thompson, 2009)	28 videos of 32 total observations	Observe Ray's *naturally occurring* coding activity during projects
	60 to 120 minutes per session	Compare against reflective data (interviews)
Observational interviews (Doheny-Farina, 1993; Katz, 2002)	28 questions derived from coding/ code during observations	Compare Ray's perceptions against my inferences
Semistructured interviews (Spradley, 1979)	Three ~90 minute interviews	Collect Ray's perception of historical coding experiences
		Compare against in situ data
Artifacts (Bracewell & Witte, 2003; Wickman, 2010)	Code and correspondence from 11 projects	Compare artifacts against in situ data
Task-based/contractive		
Retrospective accounts (Greene & Higgins, 1994)	Two screencasts of Ray recounting his coding decisions by watching previously recorded screen activity	Compare Ray's perceptions against inferences and his prior statements
Think-aloud protocols (Schriver, 1991; Swarts et al., 1984)	Five audio/video recordings of screen activity during a coding task	Compare Ray's perceptions against inferences and his prior statements
	Think-aloud protocols varied in length: 8 to 20 minutes	

prior theories about his text senses and distributions. Specifically, I reduced the data to two projects, based on available data of slicing tasks: state toxic sites (TS) and the health-texting (HT) campaign. The analysis of Ray's coding was guided by the previously discussed intermediary framework: Haas's (1996) *text sense*, Bracewell and Witte's (2003) *work ensemble*, and Clayson's (2018) *spectrum of durability*. This methodology guided my *contractive* decisions to analyze Ray's accompanying people, texts, and tools in relationships to the explicit aims versus the emergent problems.

I analyzed in situ data, such as TAPs and retrospective accounts, to gather more inferential evidence about Ray's meaning-making activity, as he planned and made coding decisions during slicing tasks, and as he constructed and overcame issues along the way. Prior writing researchers (Schriver, 1991; Swarts et al., 1984) discuss how TAPs provide rich in situ evidence for researchers to theorize writerly processes. I specifically used TAPs to explore (1) how Ray coordinated his coding ensemble to complete his slicing tasks; and (2) how he constructed and resolved any problems. Swarts et al. note that participants are of course limited in what they can verbalize, but they highlight how TAPs provide details into the sequencing of a writer's process that a researcher would otherwise only wonder about, or how a text alone lacks this more robust picture of writing (p. 53). I triangulated TAPs with other data types collected during this period of the study, such as his discourse with colleagues and observational interviews conducted after the TAP and during an observational day.

Findings

In this section, I report findings from an analysis of Ray's coding that produced data *slices*. I generated these findings from an analysis of Ray's situated coding activity that yielded results about the intermediated relationship held together by (1) his spectrum of durability, (2) how he distributed his coding across his writing ensemble, and (3) his text senses of the code and data as they developed throughout the slicing task.

Slicing Data Across a Spectrum of Durability

Ray's main coding goal during data processing and analysis tasks included the slicing of data sets into slices, which played an integral role in developing news stories. Across four projects, and the available source code within the units of processing and analyzing, Ray coded 120 total slices of data. His coding quickly combined, selected, aggregated, and computed the information from their original text into a reduced version. Slicing acts often worked

Table 3. Total Slices as Per Their Type of Durability Coded by Ray Across Four Projects and Available Data.

Project	Slices/ ephemeral	Slices/ provisional	Slices/ established	Total slices	No. of code files	Days observed
City payroll	n/a	29	0	29	3	3
Weather relief, home-rebuilding program	n/a	12	1	13	2	2
Health-texting campaign	8	27	7	42	1	3
State toxic sites	21	25	11	36	9	4
Total	29	93	18	120	13	11

Note. The shaded portions emphasize the total number of slices; n/a = not applicable.

in concert with story angles: questions and hunches that the team considered in pursuit of a more interesting story. Three main types of slices were observed to be coded by Ray, according to the following types of durability, which I revised from Clayson's (2018) original categories: *ephemeral, provisional*, and *published* (see Table 3).

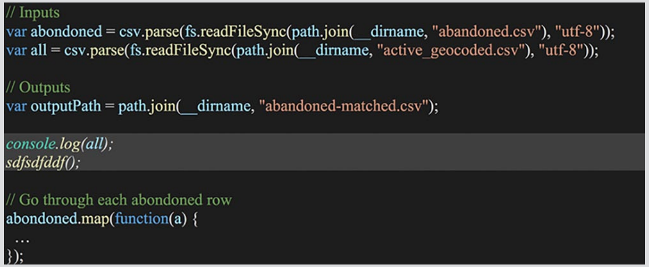
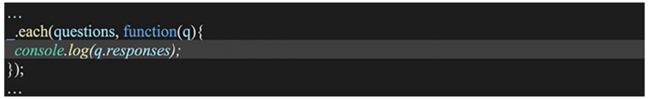
Ephemeral slices were only found during the analysis of the three situated slicing tasks, since they were either verbal representations by team members during meetings, such as angles, or they were coding acts to output slices for quick insight into an issue that cropped up during the task itself. Regarding the verbalized angles, I observed Ray and Vince transform verbalized angles into lists with a text editor or their messaging application called Slack. Ray never saved these lists, as they fulfilled a momentary need to scaffold his coding goals. While Slack could retain a record of them, only particular angles that the team refined and deemed of interest were coded into source files and documentation; hence their ephemeral status. After meetings, Ray transformed these aforementioned ephemeral representations by sometimes coding ephemeral slices in JS, which provided him the capacity to read the data as it was being drafted to code *provisional* or *established* slices. These coded ephemeral slices usually used the **console.log()** method in JS, which he either changed within seconds or deleted from the file with no trace of their production or use. Coded ephemeral slices were important for code testing, that is, Did I import the data correctly? Is this new function or conditional statement writing the data slice that I need? However, as I report in the following sections, coded ephemeral slices also enabled Ray to make sense of the integrity of the data as an ethical representation of the phenomena in question.

Provisional and *established* slices were more durable and saved within a JS file as either console logs to output data to the terminal or output a saved file in either comma-separated value (CSV) or JavaScript Object Notation (JSON) file formats. The distinguishing factor between provisional and established slices were whether or not the information from the slice was published or used in the story, since established suggests the act to establish a durable record for a wider audience. Provisional slices were preliminary in their potential narrative use and used within the team's efforts to make sense of the data and story. These slices were also either saved as console logs for output by the source code file or as files in formats such as CSV or JSON. In so doing, Ray documented major steps in data processing and analysis in both the code, which included JS functions that produced these more durable data format files, and in the organization of output files within the project directory. Essentially, Ray judged these types of provisional slices as important markers for himself and the team to trace steps back and forth between the original data and the potential published version. Based on my observations, Ray also produced an over-abundance of provisional slices, which he thought could help the team deliberate about the story angle and ultimately yield a publish-worthy version. He would output numerous files that were not necessarily reviewed in closer detail by himself or the team. According to Ray (observational interview), he considered such provisions as potentially important to be ready for alternative narrative pathways.

Out of the 120 total observed slices, only 18 slices were *established* by the team to be used in some direct fashion within the news story. In Table 4, I provide representative samples of each type of slice durability, but specifically established examples involved coding the finalized slice of information used within the published news story. For example, in established row (*est.i*), Ray's code output a JSON file with the site's ID, latitude and longitudinal coordinates, whether or not the site had an assigned manager, and whether or not the site was actively managed or not. This information, among another established slices of contextual data, were used to build an interactive map for the TS story.

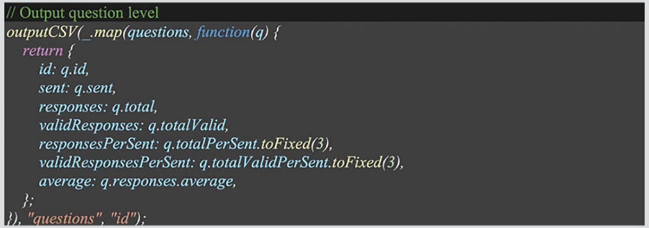
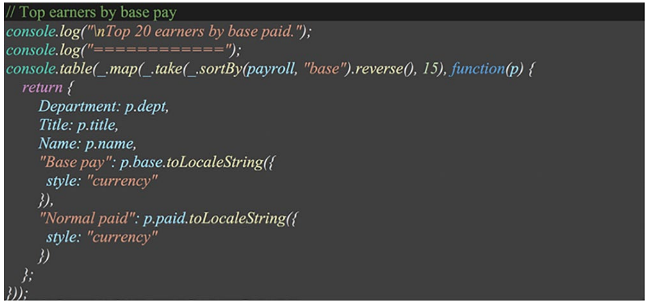
This analysis of Ray's slicing illustrates how slices are not uniformly created nor used or taken up equally. Ray coded ephemeral, provisional, and established slices for a variety of reasons and transformed data across multiple durable distributions prior to the publishing of the story. Whereas these findings focus on Ray's spectrum of durability, the next section reports findings about how Ray negotiated emergent problems by distributing this spectrum of texts across his ensemble: How does Ray's coding negotiate emergent problems by distributing texts across an ensemble of people, tools, and artifacts?

Table 4. Example Source Code Demonstrating Slice Durabilities: Ephemeral, Provisional, and Established. Bordered Source Code Signals an Analytical Code Segment, if More Code Is Quoted. Ellipses Mark Omitted Source Code for Brevity.

Durability property	News project	Example source code segment	Contextual description
Ephemeral	State toxic sites	(e.*i*)  `// Inputs` `var abondoned = csv.parse(fs.readFileSync(path.join(__dirname, "abandoned.csv"), "utf-8"));` `var all = csv.parse(fs.readFileSync(path.join(__dirname, "active_geocoded.csv"), "utf-8"));` `// Outputs` `var outputPath = path.join(__dirname, "abandoned-matched.csv");` `console.log(all);` `sdfsdfddf();` `// Go through each abondoned row` `abondoned.map(function(a) {` `...` `});`	(e.*i*) Ray wrote the log output to test if the *active_geocoded.csv* file had been imported and read the available information. The *sdfsdfddf*() function is superfluous and only meant to error-out the execution of the file after the log output, so he could avoid other outputs conducted after this log. He initially noticed possible ID issues with the data sets, due to this log output. He subsequently deleted it from the file to pursue this problem.
Ephemeral	Health texting	(e.*ii*)  `...` `_.each(questions, function(q){` `console.log(q.responses);` `});` `...`	(e.*ii*) Ray output the questions data to verify that his code is creating the desired structure and content. He changed the log input and function code shortly thereafter to verify these code changes. Then, when completed, he deleted the log.

(continued)

Table 4. (continued)

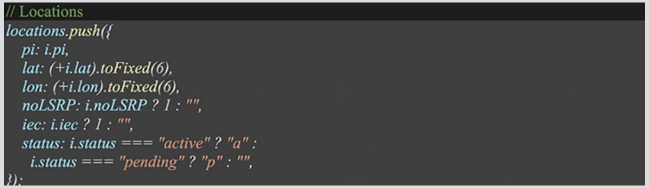
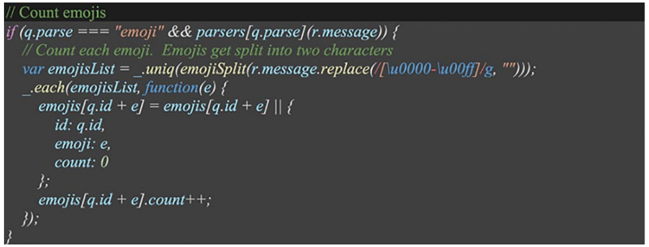
Durability property	News project	Example source code segment	Contextual description
Provisional	Health texting	(*p.i*) 	(*p.i*) Ray takes the questions data object and slices it based on Vince's angles: "For every day . . . % responded More/Less/Same by goal . . . 1 to 5 scale average by goal . . ." (Vince, Slack message). This question-level slice helped him refine slices to use in the reported story.
Provisional	City payroll	(*p.ii*) 	(*p.ii*) Ray outputs a table of values to the terminal based on the *per person* index as cross-referenced with the base-pay value. He copies and pastes the terminal output with the team as a message on Slack. The slice helped the team refine their angles to produce a better potential story.

```
// Output question level
outputCSV(_.map(questions, function(q) {
  return {
    id: q.id,
    sent: q.sent,
    responses: q.total,
    validResponses: q.totalValid,
    responsesPerSent: q.totalPerSent.toFixed(3),
    validResponsesPerSent: q.totalValidPerSent.toFixed(3),
    average: q.responses.average,
  };
}), "questions", "id");
```

```
// Top earners by base pay
console.log("\nTop 20 earners by base paid.");
console.log("============");
console.table(_.map(_.take(_.sortBy(payroll, "base").reverse(), 15), function(p) {
  return {
    Department: p.dept,
    Title: p.title,
    Name: p.name,
    "Base pay": p.base.toLocaleString({
     style: "currency"
    }),
    "Normal paid": p.paid.toLocaleString({
     style: "currency"
    })
  };
}));
```

(continued)

Table 4. (continued)

Durability property	News project	Example source code segment	Contextual description
Established	State toxic sites	(*est.i*) 	(*est.ii*) Ray codes location details that he outputs as a JSON file later in the file, which become the variables rendered on the interactive map interface.
Established	Health texting	(*est.ii*) 	(*est.ii*) Ray outputs a CSV aggregated by *per goal* and eventually used it to create a chart of how many emojis users used *per goal*.

```
// Locations
locations.push({
  pi: i.pi,
  lat: (+i.lat).toFixed(6),
  lon: (+i.lon).toFixed(6),
  noLSRP: i.noLSRP ? 1 : "",
  iec: i.iec ? 1 : "",
  status: i.status === "active" ? "a" :
    i.status === "pending" ? "p" : "",
});
```

```
// Count emojis
if (q.parse === "emoji" && parsers[q.parse](r.message)) {
  // Count each emoji.  Emojis get split into two characters
  var emojisList = _.uniq(emojiSplit(r.message.replace(/[\u0000-\u00ff]/g, "")));
  _.each(emojisList, function(e) {
    emojis[q.id + e] = emojis[q.id + e] || {
      id: q.id,
      emoji: e,
      count: 0
    };
    emojis[q.id + e].count++;
  });
}
```

Note. The shaded portions differentiate the excerpts of code; CSV = comma separated value; or JSON = JavaScript Object Notation.

Table 5. Co-occurrence Results From an Analysis of Ray's Situated Coding Acts Across his Coding Ensemble During Three Slicing Tasks: Dimensionalized by Explicit Aims and Emergent Problems.

Coding ensemble technology	Explicit aims	Emergent problems	Total coding acts
Atom	92	124	216
Terminal	18	40	58
Calc	6	40	46
Slack	11	16	27
Chrome	0	9	9
Total coding acts	127	229	356

Note. The shaded portions emphasize the total number of coding acts.

Ray's Coding Ensemble and Data Distribution Across Explicit Aims and Emergent Problems

In this section, I analyze of three in situ slicing tasks: one during the TS project, and two during the HT project. Across these tasks, findings indicate that Ray devoted more coding acts to emergent problems (229/356 coding acts), rather than explicit slicing aims (127/356 coding acts; see Table 5). Coding acts were defined by how Ray coordinated each coding technology to complete a JS statement or expression (see the glossary of terms in Appendix A). For example, if Ray instantiated a variable to tally the number of TS ID matches across two data sets, '`var matches;`' this constituted a singular coding act in the service of the explicit aim. If Ray wrote a conditional operator statement that parsed the data set and searched for discrepancies with site IDs, this moment would involve at least three ensembled acts to handle an emergent problem: (1) reading the data set in Calc spreadsheet program, (2) coding the conditional statement to output site ID numbers to the Terminal, and (3) running the script and reading its ID output to verify the problem.

During each task analyzed, every technology had some general functions. Across all tasks, Ray used his code editor (Atom), the Terminal, the spreadsheet application (OpenOffice Calc), and Slack (messaging application) to complete his coding tasks.[12] He conducted the majority of his coding in Atom (216/356), followed by his coordinated use of the Terminal (58/356), Calc (46/356), Slack (27/356), and Chrome web browser (9/356). Atom and the Terminal were integral for Ray's slicing, because they enabled him to write JS and "print out" the data if he deemed it useful to understand how his code transformed the data.

Ray often used Atom, Calc, and the Terminal in tandem to help him overcome emergent problems with both the data sets and code. As reported before, he wrote **console.log()** statements in Atom, so he could run the code to print out data slices to the Terminal. These distributions of digital data into readable texts helped him read to understand how the Node.js compiler-interpreter[13] had transformed the data (or not). If Atom and the Terminal helped Ray read the data as reduced segments, Calc helped Ray read and search the data set in its original entirety. For example, a bulk of his acts in Calc occurred in the *emergent problem* category (40/46). In some cases, he used Calc's text search feature, so he could quickly search the list for specific data properties. He used Slack and Chrome to handle issues beyond his personal understanding. He used Chrome twice to reference the documentation about a particular code-library's computational method,[14] such as Lodash's **_.findWhere()**, so he could clarify its syntax and data parameters.

Ray's ensembled coding differed across the two projects analyzed in the present study largely due to how the provenance of the original data differed greatly. In the TS project, seven data sets were used from the dozens of collected sets from a State Department agency via its website and a Freedom of Information request, while the HT project used data collected by a web-texting application that Ray had developed himself and pulled from a database directly. Since Ray had worked extensively with the HT data, prior to these initial slicing tasks, he already had historical experiences with its provenance, content, and structure. Additionally, before his meeting with the journalism team on the HT project, Ray had already been slicing the data based on his own educated guesses about what they might want to investigate. On the TS project, however, Ray had never looked at the data prior to the task. Consequently, Ray knew the HT data far more extensively than the TS data, and he had already written some code that processed the data from the original HT database. This difference had consequences that manifested in several ways. For example, if the co-occurrence findings are dimensionalized by project (see Table 6), tallies highlight how Ray distributed far more emergent problems across his initial TS slicing task versus the two HT slicing tasks.

In this section and the previous one, results show how Ray distributed code and data across an ensemble of people, texts, and tools, which helped him develop and resolve emergent problems with his slicing. In the following section, I analyze the text sense dimensions of Ray's textual distributions across his ensemble during two of the three tasks to infer some of the accompanying development of Ray's senses of the texts—decisions that his code did not necessarily document for analysis.

Table 6. Co-Occurrence Results of Ray's Situated Distribution Acts: Dimensionalized by Project.

Project (total tasks)	Atom	Terminal	Calc	Slack	Chrome	Total coding acts
Toxic sites (1)						
Explicit aim	85	9	5	0	0	**99**
Emergent problem	48	28	37	16	5	**134**
Health texting (2)						
Explicit aim	44	9	1	11	0	**65**
Emergent problem	39	12	3	0	4	**58**
Total coding acts	**216**	**58**	**46**	**27**	**9**	**356**

Note. The shaded portions emphasize the contrasting co-occurence tallies of distribution acts observed during each project's emergent problems.

Ray's Coding Senses: Data, Programmatic, Contextual, and Historical Senses

In this section, I report findings about the range of Ray's text senses in relation to a range of ensembled distributions, which is guided by the third question: What sense of the code and data as texts can be inferred from his meaning making work with his coding ensemble? Haas (1996) originally defined text sense in relationship to a writer's sense of the finalized material text. Clayson (2018) theorized how text sense develops across a range of distributed representations of the text by focusing on the spectrum of durability produced by writers to support this meaning making. Smagorinsky (2001) noted how sense often involves a capacious degree of conceptual associations as a literacy act unfolds. Due to the complex of sense associations that writers develop across a writing task, I extend the prevailing definition of text sense beyond the materialized goal alone. I report four text sense properties specific to Ray's coding as an intermediary writing of code and data that I call *coding senses*. Coding senses refer to any ensembled coding act that signified insight into the following properties that were not and/or could not necessarily be documented within the texts he produced: data sense, programmatic sense, contextual sense, and historical sense (see Table 7).

Each section below elaborates on these definitions by first establishing Ray's coding situation that describes the events that led up to the analyzed slicing task. Then, I describe the relationships between Ray's coding senses and spectrum of textual distributions during a slicing task.

Task 1: State toxic sites: Transposing data and investigating data set integrity. Ray's initial slicing task on the TS project was simple: match up TSs across

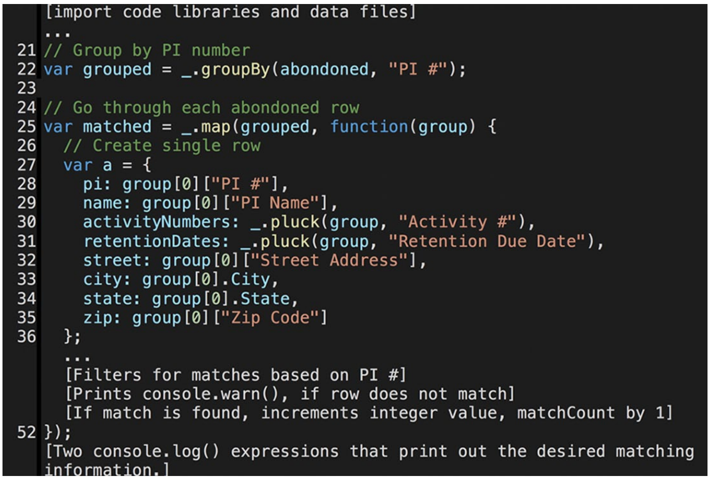
Table 7. Definitions of Ray's Coding Senses Inferred From Two Slicing Tasks.

Sense property	Definition
Data	Any ensembled act that signifies insight into the output data
Programmatic	Any ensembled act that signifies insight into how the code will programmatically transform the data by either transposing, selecting, computing, or aggregating it
Contextual	Any ensembled act that signifies questions, comments, or insights about the context of the task
Historical	Any ensembled act that signifies questions, comments, or insights about the task in relationship to prior experiences

two files. The matching analysis compared a set of "active" sites, which included some desired and accurate geocoded values (latitude and longitudinal coordinates) with a set of "abandoned" sites without these values. The explicit aim: How many more sites need to be geocoded with the future goal to use those coordinate values for an interactive map? Despite this simplicity, note how ID issues accrued into 99/134 total emergent problems distributed across his coding ensemble. Ray's understanding of the ID problem changed as he coded outputs of each row as ephemeral slices in the Terminal, where he subsequently perceived the problem and reviewed the data sets more thoroughly in Calc. During the task, he learned how the rows with "duplicate" (Ray, Slack message to Jun) IDs contained "different" or "changing" column values for the same site. His insights resulted in the code excerpted in Figure 1, where lines 22, 30, and 31 in particular account for emergent ID issues (see Appendix B for an extended excerpt.)

These code segments are textual distributions of Ray's inquiries and responses about the data sets and project. On lines 22 and 25 to 36, Ray arrived at two key ways to "line up" the two sets, so he could also complete the matching task. Ray's initial code simply attempted to use an `if` conditional statement to count ID matches, but the task warranted more processing work to complete this simple sum. This alignment writing transposed the "abandoned" CSV data set to align it with the structure of the "active" CSV set. In sum, he wanted both sets to be indexed in the same way, prior to filtering out matched rows to count.

On line 22, Ray had read the data in both the Terminal and Calc, where he learned that the two sets had the same PI number that served as a unique ID. However, they were not uniform. Some of the ID numbers in the abandoned set had a leading G and leading zeroes, which he learned were vestiges from older database systems. Furthermore, the abandoned set was not organized by the PI number, since the column was not listed as the first column name in the



```
   [import code libraries and data files]
   ...
21 // Group by PI number
22 var grouped = _.groupBy(abondoned, "PI #");
23
24 // Go through each abondoned row
25 var matched = _.map(grouped, function(group) {
26   // Create single row
27   var a = {
28     pi: group[0]["PI #"],
29     name: group[0]["PI Name"],
30     activityNumbers: _.pluck(group, "Activity #"),
31     retentionDates: _.pluck(group, "Retention Due Date"),
32     street: group[0]["Street Address"],
33     city: group[0].City,
34     state: group[0].State,
35     zip: group[0]["Zip Code"]
36   };
     ...
     [Filters for matches based on PI #]
     [Prints console.warn(), if row does not match]
     [If match is found, increments integer value, matchCount by 1]
52 });
   [Two console.log() expressions that print out the desired matching
   information.]
```

Figure 1. Excerpt of Ray's JavaScript code, which he was developing during the toxic sites project. Line numbers match the original file, as a means to indicate excluded elements noted within the square brackets.

CSV file. For these reasons, Ray wrote the **_.groupBy()** function that typically accepts two main arguments: a collection (**abondoned** [sic]), and an iteratee (**"PI #"**), as per the Lodash (Version 3.8.1) documentation (Sirois & Hall, 2015). The **groupBy()** method creates a new object array, which he assigned to the **grouped** variable, by iterating through all elements within **abondoned** [sic] collection. If the two sets had been more consistent, then Ray would not have needed to code these excerpted parts to complete his matching task. Due to these inconsistencies, Ray coded the script to rewrite every row with some processing techniques (lines 27-36). In what follows, I report the different coding senses that I was able to infer from multiple sources about Ray's decision to develop the code in Figure 1.

Contextual sense. Recall how this task was his first on the TS project. He began it by reading the data sets in Calc and messaging Jun about the "context" (Ray, observational interview) of the data for the first 15 minutes. Ray continued to message Jun as he began to code the matching function, asking her questions based on new information about the data. He asked Jun questions about which files were important, since, at the time, the project folder had over 20 data set files with no documented organization, file-naming scheme, or notes about which data sets were originally provided by the state versus those that Jun had already started to process. In an observational interview, just after his initial messages to Jun about the data, he noted how he occasionally felt as though he had "no idea about what is going on with the projects." He remarked how reporters sometimes did not document their data work, which often made his onboarding more cumbersome, since he sometimes came onto projects "after the data has been gathered and looked at [processed] a little bit." Before returning to the task, he said that he appreciated any "context" of the data and story: a small "step back to describe it a little more [the data and project]" helps him more quickly conduct his coding.

From there, Ray toggled in-between Atom, the Terminal, and Calc dozens of times (refer back to the ensembled results in Tables 5 and 6). Throughout this task, Ray had printed out the data to the Terminal (ephemeral slices) and also read it in Calc as he coded this matching slice. In-between these ensembled texts, Ray posed numerous questions to Jun about possible "duplicate" or "repeating" rows of information about the sites. To shore up more context about this data problem, Ray and Jun deliberated about the matter throughout much of the task, which overall used two distinct forms of questions: *to clarify* (132 instances) or *to verify* (14 instances) information about the data set(s). For example, some clarification interactions pertained to the provenance of the data: "Is the data from census reporter?." The repeating ID issue blended questions about the original purpose of a particular column with questions of its integrity. For example, Ray asked about the "retention due date" column: "It looks like there are some 'duplicates' in the abondoned (sic) data. They are the same but have different 'retention due date' (sic). Any insight on this?" (Slack message). In total, Ray and Jun's deliberation about the TS data sets involved 67 deliberative instances. The highest number of coded clarification interactions about the data integrity involved such questions about any perceived *value discrepancies* (21) that were linked to the repeating ID values. Any suspect discrepancies within the column values were questioned by Ray, and discussed with Jun, so they could ensure that the data were accurate, not out of date, and not obscuring the main goal of Ray's task. This process exacted an accurate total of site matches across two lists, as well as identifying any larger issues with the data and its impact on the developing story.

Programmatic and data senses. Ray's clarification and verification questions did not simply exist in this deliberation in Slack. Ray weaved their deliberation to clarify and verify the integrity of the data with his coding across his editor, the terminal, and Calc to match up the site lists. Specifically, when he wrote the code that transposed the repeating ID rows by bundling the potentially different information across the two columns into embedded array lists (Figure 1, lines 30-31). In a retrospective account, which asked him to account for his ensembled distributions across Atom, the Terminal, and Calc as he wrote this code segment, he was "going through each column [variable] and saying, 'What kind of data is it?.'" For example, he noted questions, such as Is the column a Boolean (yes/no), a date in need of formatting, or in this situation a subset list of updated dates, based on the status of the case manager (summarized responses from Ray). He elaborated that he reviewed data in the Terminal and Calc to "*visually see it* [the data set] and then *code it into an array*, if it needed to be" (emphasis added). This last item, regarding the embedded use of Lodash's `_.pluck()` method to "code it into an array," highlights how such coding senses of the context blended with senses about how to programmatically rewrite the data amenable to his quick matching task needs. Indeed, in his second of two uses of the Chrome web browser, he searched the Lodash website for Lodash's `.pluck()` method by name, reviewed its arguments for a moment, then coded them in Atom to account for the different information for any already existing ID from the abandoned set. In sum, his sense of the data—What types and structures will work for this situation?—were coordinated closely with his programmatic sense about what the computational methods would do with the data.

Historical senses. Overall, Ray's coding during this matching task highlights his experience working with data types, structures, formats, as well as the JS language and its code libraries that help him write and compute with such data. Across the task, he only consulted documentation about how to code the data twice, and both times he already knew the names of the methods before consulting it. Additionally, he verbalized his insights about how prior experiences have taught him that if he encounters a CSV with repeating IDs, but different or changing values across rows, it warrants investigation and coding that handles it, as he put it, "appropriately" (observational interview). By "appropriately," he reflected how changing information could prove either important or not, but data processing was supposed to act as documentation of all data changes from original to the data ultimately used in the story. He summed this main objective as coding for himself "six months from now" (observational interview), knowing that at any time he could be asked to pull out, look up, and answer how they arrived at their results. Additionally, after the retrospective account in an observational interview, Ray also provided a high-level description about the objective of this task:

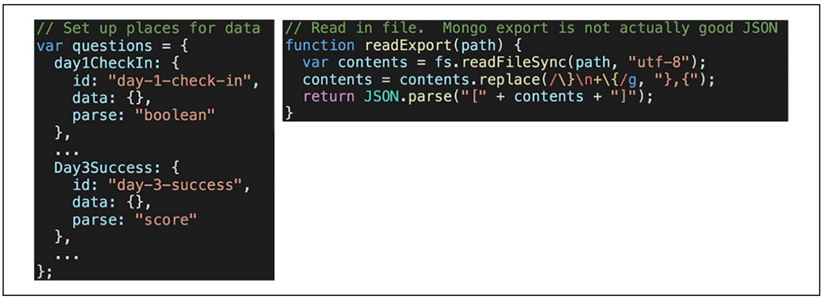
> I'm basically trying to *reverse-engineer the data*. And we have it in this data set, but it's not in the structure that it's ultimately stored in, or processed. What's probable is that there's a Site table [in SQL[15]], then there's multiple Activities per Site, and that Activity it may, you know, you've got your Activity

> Type, and maybe the Case Manager is for the Activity, so each Activity has a specific Case Manager for it. But it's hard to tell, because this "Source Information" [value] doesn't change, but that could be Source-based for each Activity. We don't know how that fits into it exactly. (emphasis added)

I do not claim that Ray was conscious of his speculation about the original structure and format of the data during the matching task. However, his verbalizing highlights his historical experiences working with data across coding domains, and how he understands that database exports flatten Structured Query Language's (SQL) relational and hierarchical tables, when transformed into a CSV format. In a semistructured interview, he remarked how CSV files are often referred to as "flat" files, due to their two-dimensional property. Furthermore, as per a different semistructured interview that focused on his coding history, he noted how he started out as a database engineer for a large organization, which involved lots of SQL database designing and querying. While difficult to trace, Ray's historical experiences coding with data afforded him the capacity to quickly slice the data, question its integrity, and provide the team with the desired matching count information.

Task 2: Health-texting campaign: Coding "one niblet of data." In this situation, Ray knew the data very well and had immediate access to it, since he designed the database and mobile-texting application that collected the data. He developed an interactive, mobile-texting application that guided over 20,000 people through a week of intentional goals to decrease their technology use, which collected response data. This section describes Ray's initial meeting with Vince, the data-team editor, and the journalist team who asked Ray to create the app. During the meeting, they brainstormed what angles to focus on for the reported results. Then, I focus on Ray's transition from the meeting and list of angles to his initial few coding acts in his code editor.

Prior to the meeting, Ray stated that he had already "pulled out" (observational video) the exported data from the texting app's database. In Figure 2 (right), Ray wrote code that imported a "Mongo export," which looks similar to JSON, but Ray noted how his import function modified and transposed certain parts of the data before coding it further in his file. In Figure 2 (left), he also created a data structure at the top of the file, which he names as his "places for the data." These "places" later enabled him to organize the imported data in the original *per Listener* structure into a *per Question* structure. Later, he assigned tallies of each *per Question* response within the `data: {}` portion. Overall, Ray had much more primary experience with this data set, being the lone "engineer," as he put it, of the texting-app and now processing and analysis code. In this section, I examine Ray's initial response to the team meeting that resulted in a list of angles for Ray to code.



```
// Set up places for data
var questions = {
  day1CheckIn: {
    id: "day-1-check-in",
    data: {},
    parse: "boolean"
  },
  ...
  Day3Success: {
    id: "day-3-success",
    data: {},
    parse: "score"
  },
  ...
};
```

```
// Read in file.  Mongo export is not actually good JSON
function readExport(path) {
  var contents = fs.readFileSync(path, "utf-8");
  contents = contents.replace(/\}\n+\{/g, "},{");
  return JSON.parse("[" + contents + "]");
}
```

Figure 2. Example code from Ray's initial code file, prior to the meeting with the steam.

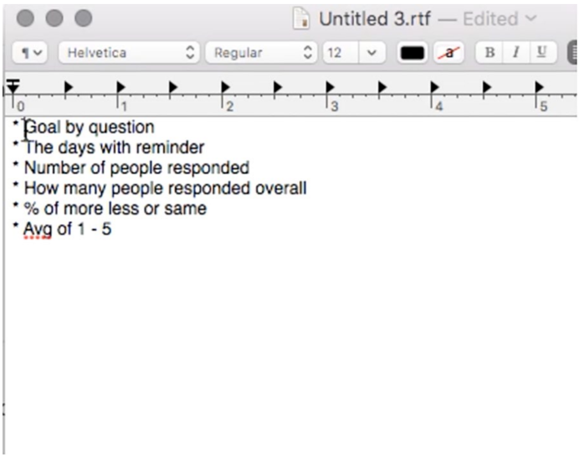
As soon as the video-meeting began, one person from the team noted their initial hunches—mainly directed at Vince. (Recall how Ray telecommutes.) They were immediately curious about "one niblet of data" (Fieldnote), as they put it. "I think there's a story there," they led on. This "niblet" was an ordinal question that asked people to rate their overall feeling of being overwhelmed or not regarding their technology use for the day on a scale of 1 to 5. Amid the quick interchanges between the team and Vince, Ray listed their possible angles to pursue with the data in a text file (see Figure 3). After about 10 minutes, the meeting adjourned, and Ray shifted his attention to his list.

"You gotta list going there?" I asked Ray (video recording). "Yeah, but I didn't actually . . ." His attention diverted back to his laptop, and he navigated to Vince's Slack messaging channel and wrote, "Can you send me your notes from that [meeting]? I didn't take very good ones" (observational video).

Vince responded with a stream of items that he neatly organized by types of "Desired data" (observational video; see Figure 4, left). While Vince listed off angles, Ray jotted down two more angles of his own (see Figure 4, right), including the requested "niblet." He reviewed with me how the team were curious about the percentage of people who noted feeling less overwhelmed by technology after the first day. This angle was a hopeful talking point for the show being recorded later that afternoon, while the remaining results would be reviewed and used for a special report with visualizations the following week.

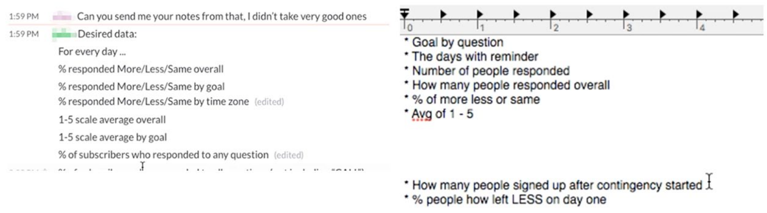
Ray took a moment to review Vince's list. I asked him what he was thinking. His response interwove coding senses worth quoting at some length:

> I'm still trying to get it [Vince's list] logically in my head. But this, [*he hovers his mouse across the "For every day . . ." section* (see Figure 4)], is all for each day. There's gotta be a way to go through it all [the question-level data] and


Untitled 3.rtf — Edited

- Goal by question
- The days with reminder
- Number of people responded
- How many people responded overall
- % of more less or same
- Avg of 1 - 5

Figure 3. Ray's list of angles from his meeting with the journalism team (screen capture of observational video).


1:59 PM Can you send me your notes from that, I didn't take very good ones
1:59 PM Desired data:
For every day ...
% responded More/Less/Same overall
% responded More/Less/Same by goal
% responded More/Less/Same by time zone (edited)
1-5 scale average overall
1-5 scale average by goal
% of subscribers who responded to any question (edited)

- Goal by question
- The days with reminder
- Number of people responded
- How many people responded overall
- % of more less or same
- Avg of 1 - 5

- How many people signed up after contingency started
- % people how left LESS on day one

Figure 4. Partial capture of Vince's list of angles from the team meeting (left), and Ray's list with additional "niblet" angle (right; screen capture of observation video).

create all of it [each angle at the day-level] and not do each one [angle] in particular. We've got it [the listener response data] broken down into each day [daily question already in the existing code file]. . . . [*Pauses*]. We have these questions [already written in the code] (see Figure 2, left), so my biggest problem is given that we have this [per Question] structure, [*navigates to his code editor and scrolls up to a JS object array in results.js file*], and per Day prompts [*Pauses*] . . . I know this doesn't answer all of these questions. [*Gestures back to Vince's list in Slack, but returns quickly to the code editor*]. But, we know for day 4, [*gestures to line 90 in the results.js*], the success question is the data-type score, so it's that 1 to 5 question. I can just look

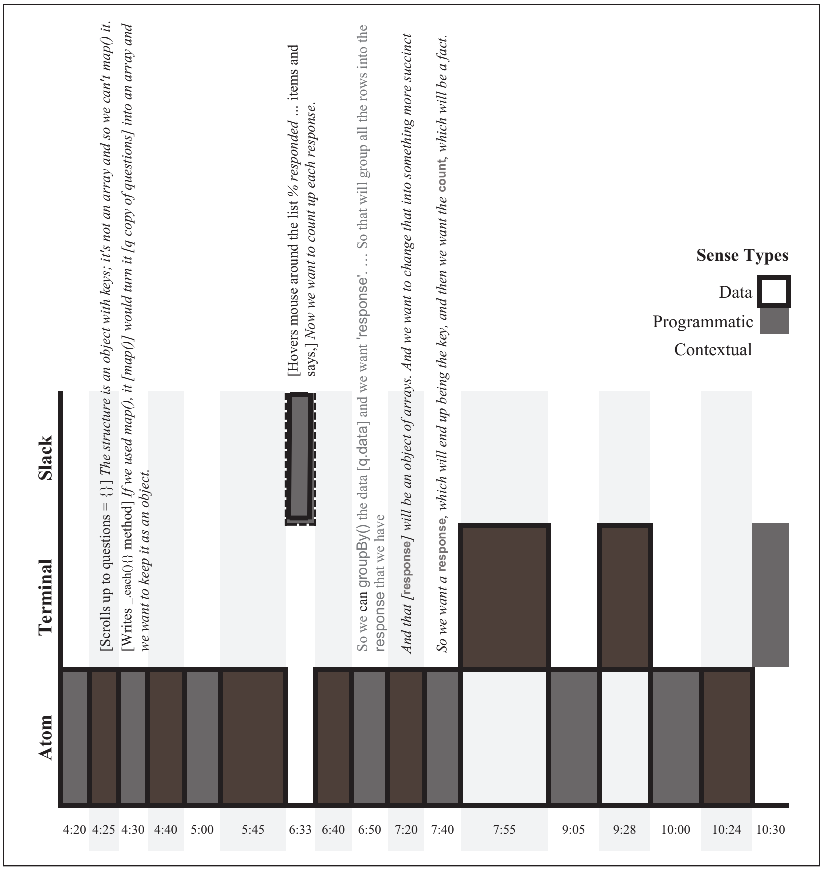
> through here [`data: {}`] (see Figure 2) and pull out this information: these aggregate informations [sic]. Then, I'll have a fairly big data structure that basically has all of [*gestures back to the list in Slack*] these questions [angles]. [*Pauses*]. I don't know the best way to output them in a way that makes sense, but that's the best that I can do. (Observational video)

After this moment, his coding did much of what he articulated above. He coded a function within which he could isolate three of the listed angles in under 20 minutes. He wrote a JS function that incorporated some computational methods that would "go through" the exported data from the original database, create a "fairly big data structure," so he could output "all of it [the desired angles and the day-level data]" within this additional function in the existing JS file. In what follows, I describe the coding senses inferred after this moment, as he coded the slices jotted down in his list.

Programmatic and data senses. In the moment above, note how Ray verbalized programmatic and data senses. He articulates a *data sense* by describing `questions`' existing structure and content, which scaffolded his sense about how to code `questions` programmatically into his desired slices that respond to Vince's list of angles. His *data sense was intermediated by his programmatic sense* about how his existing code can potentially transform the data.

In Figure 5, I plotted the initial approximately 6 minutes of Ray's slicing task, which occurred just after his above verbalization. I created it by cross-referencing his coding senses inferred from think-aloud data and this prior verbalization of the task. He used his sense of `questions` as an object with the desired information to choose his programmatic methods. Since `questions` was an object—not an array of objects—he noted how he knew that he could not use Lodash's `_.map()` method, so he used it's `_.each()` method instead. Furthermore, he noted the consequences of using `_.map()` with `questions`, since it would change it into an array. He did not want to do so, since he was planning on using this segment of the code—demarcated by the comment "`// Create aggregate data`"—across other angles.

Figure 5 highlights Ray's intermediary relationship between data and his code that performs on data, where the two forms of senses for this task were tightly bound. In addition to his knowledge about the consequences of using either `_.each()` or `_.map()`, his repeated use of the `_.each()` method (see Figure 6, lines 308, 310, and 315) demonstrates how he knew that he needed to "go through" (Ray in aforementioned observational interview), that is, traverse, the object's nested hierarchical structure to arrive at `questions`' `data: {}` "places" (Figure 2, left) to ultimately compute comparative sums (see Figure 6, lines 317 and 329) between valid and invalid response data.



[Scrolls up to questions = {}] *The structure is an object with keys; it's not an array and so we can't map() it.*

[Writes _.each(){} method] *If we used map(), it [map()] would turn it [q copy of questions] into an array and we want to keep it as an object.*

[Hovers mouse around the list *% responded* ... items and says,] *Now we want to count up each response.*

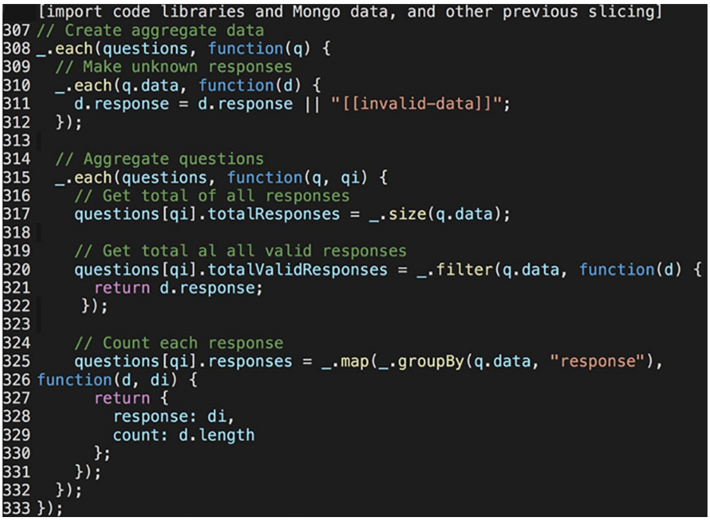
So we can groupBy() the data [q.data] and we want 'response'. ... So that will group all the rows into the response that we have

And that [response] will be an object of arrays. And we want to change that into something more succinct

So we want a response, which will end up being the key, and then we want the count, which will be a fact.

Figure 5. Temporal chart of Ray's distributions during an approximately 6-minute excerpt from a slicing task on the health-texting project (think-aloud protocol). One coding technology is analytically coded per timestamp (Atom, Terminal, or Slack), but multiple coding senses can occur per instance (data, programmatic, contextual). Ray verbalized for each timestamp, but I only included the above segments to highlight a notable moment.

Additionally, his foresight to filter out invalid responses with a String textual marker, **"[[invalid-data]],"** and then **_.map()** the filtered **response** data by using the **_.groupBy()** method (Figure 6, line 325) also further demonstrates his intermediated sense of the code and data in their joint textual development. This point is further supported by the fact that he coded the excerpted code above (Figure 6) without any reference to ephemeral



```
    [import code libraries and Mongo data, and other previous slicing]
307 // Create aggregate data
308 _.each(questions, function(q) {
309   // Make unknown responses
310   _.each(q.data, function(d) {
311     d.response = d.response || "[[invalid-data]]";
312   });
313
314   // Aggregate questions
315   _.each(questions, function(q, qi) {
316     // Get total of all responses
317     questions[qi].totalResponses = _.size(q.data);
318
319     // Get total al all valid responses
320     questions[qi].totalValidResponses = _.filter(q.data, function(d) {
321       return d.response;
322      });
323
324     // Count each response
325     questions[qi].responses = _.map(_.groupBy(q.data, "response"),
326 function(d, di) {
327       return {
328         response: di,
329         count: d.length
330       };
331     });
332   });
333 });
```

Figure 6. Excerpt of Ray's JavaScript code written during the health-texting project. Line numbers match the original file, as a means to indicate excluded elements noted within the square brackets.

slices in the Terminal. It was only after this initial 6-minute period that he decided to print out the data to the Terminal to verify if his code was indeed acting on his desired behalf of the team. This intermediation of programmatic and data senses was also evidenced by his prediction that his first ephemeral slice printed to the Terminal would most likely not provide the details he wanted to review. As he wrote his first log statement, he noted,

> So just to make sure we're not totally screwing stuff up, we're just outputting it to the console, so we can see the output in the command line, but I'm not sure if it's going to be nested enough to see what we want to see. Node's console statement doesn't necessarily . . . yeah, it's not there.

Overall, what code he wrote was intermediary to what data he wanted to write and read in the Terminal.

Contextual senses. Ray's *contextual sense* was intermediated by the team's ephemeral verbalization of their "niblet" angle and Vince's list of this angle and others. This ephemeral slice was based on their hunch—a particular value that they deemed worthy of reporting, if it yielded results that they considered interesting. It also arguably scaffolded his subsequent coding reported above. Ray only referred to the list once during this initial 6 minutes, and he did so to recall the angle of interest at the moment where he coded the slice: "Now we want to count up each response." Again, note how his verbalization—"response" and "count"—resulted in becoming the key and fact for the future established slice (Figure 6, lines 328-329).

Historical senses. His *historical sense* can be inferred from his capacity to write Node.js forms of JS code with the Lodash code library. In this situation, he only consulted the Lodash documentation once in relationship to an issue with his code that filters the invalid responses. Yet, even then, he knew the name of the method (**_.reduce()**) beforehand, indicating his prior experiences. Additionally, Ray also had a sense of the nested hierarchical structure of the exported data and its subsequent placement and structure as the **questions** object.

Discussion and Implications

Among the hundreds of slices that Ray coded during this study and dozens that I observed him produce in situ, his slicing demonstrated an intermediary relationship between his ensemble and the range of durable representations that he produced throughout each task to develop his coding senses of the texts. From discussions and lists of angles, Ray read and wrote code and data jointly in Atom and the Terminal. He also read data within Calc and consulted his colleagues on Slack about emergent data problems or his slicing goals. He occasionally consulted documentation online, when handling more minor coding syntax issues. Multiple conclusions can be drawn from findings about how he coordinated this ensemble.

Clayson (2018) proposed three main forms of distribution on a spectrum of durability: provisional, persistent, and permanent representations (p. 167). For Clayson, *provisional* included gestures with no traceable material form beyond its original expression. She defined *persistent* as a material form that is never used within the final document, whereas *permanent* distributions were deemed appropriate for the audience. In my analysis, slices and their production are categorized with the following revised terms: *ephemeral, provisional*, or *established*.

I propose these revisions for the following reasons. If Clayson's initial *provisional* category is renamed as *ephemeral*, it can draw attention to distributions beyond gesture. Ephemeral describes the fleeting temporal aspect of certain sense-making distributions, rather than the gestural-only found in Clayson's study. Ephemeral also diversifies the range of more fleeting distributions that may render a text, but only momentarily, as evinced by Ray's slicing with momentary uses of the `console.log()`. I also suggest replacing *persistent* with *provisional*, because while distributions can certainly be more persistent than others, provisional captures this persistent property *and* the intermediated sense-making linked with their use. Indeed, for Clayson's report-writers, lists on a whiteboard distributed provisional segments of the report, while documented slices in Ray's code as log outputs or as saved data files offered Ray and the team provisional slices about the developing aggregate dimensions of the story. Additionally, Clayson coined the third as *permanent*, which suggests that the text become fixed and ready for their audience. I suggest the use of the term *established*, because permanent suggests just that something out there forever. Established draws attention to the intermediated labor involved in establishing a text sense as a *configuration of signs* (Smagorinsky, 2001; Witte, 1992) deemed ready for audiences—even if an audience includes a computational system designed to render an interactive map. I argue that these changes account for a wider range of forms of writing, including coding, and also foregrounds the intermediated relationships with a writer and their ensemble.

Additionally, Ray's ensemble and accompanying coding senses supported decisions about what contextual and historical factors mattered, as he constructed and negotiated emergent problems throughout his slicing tasks. In a data-journalism domain, Ray and his colleagues encountered data sets from myriad sources with prior purposes, forms, contexts, and content beyond their initial understanding. This case examined the role of the often-elided data-processing work to contextualize the data by illustrating Ray's coding as rewriting data for new goals and purposes. Ray's slicing involved much more than transposing skills, since context came to matter for Ray in surprising ways. Recall how contextual factors about the data and his coding changed across projects analyzed above. Projects like TS involved much more effort on Ray's part to understand data provenance in relationship with team's new goals for it. Conversely, Ray had internalized a sense of the structure, content, and purposes linked to the HT data, since he originally developed the app that collected and stored this information for a team within the organization. These findings suggest that data-processing and database design may play a more complicated meaning-making role than the current perception shared across data-science domains.[16] These findings also affirm recent studies of software systems from the social sciences and humanities (Benjamin, 2019; Bucher, 2018; Noble, 2018; Sano-Franchini, 2018), which highlight the consequences of neglecting historical, social, material, and situational factors from the perspective of users. From these findings, future studies could investigate more closely any patterns about what aspects of context come to matter for the developers and designers of these software systems: How and when tacit historical and contextual considerations come to matter, or what assumptions are tacitly operating and in need of more critical examination. Writing studies offers new research avenues to examine how developers as writers are limited by their own capacities to perceive and resolve problems with their code beyond the narrow technological perspective.

There are important limitations to this study to consider as well. First, this case offers a substantive theory of coding as an intermediary form of writing, rather than a generalizable one. This limitation, however, is also a possible strength, since a grounded investigation of coding with an unusual case methodology had yet to have been conducted. The aims of this case study have been to *start* the theorizing process and construct new lines of inquiry to explore and theorize coding as an intermediary form of writing. Second, I cannot provide intercoder reliability, due to my agreements with Ray's professional organization. I mitigated this constraint by triangulating multiple types of data sources and checks with Ray that I conducted during observational interviews. Additionally, findings linked to both ensembled distributions and senses were also triangulated across the multiple data-types: combinations of TAPs and screen-recorded activity with interviews, fieldnote observations, and retrospective accounts. Future researchers can test the veracity and boundaries of my proposed terminology across similar or different domains of writing, which should include computer coding. Future studies could also compare slicing activity across data journalism and other data science contexts, and verify similar instantiations of slicing activity to valorize and enrich explanations about data-processing and analysis.

Future studies would do well to examine coding as an intermediary form of writing with data, where code is not a discrete technological object, but rather the dynamic result of ensembled writing activities that coordinate people, texts, and tools. This substantive finding should open up further inquiry that bolsters the premise that computer code cannot be reduced to its linguistic sign or its voltaic-registered sign. For Ray, code did not exist as a discrete 1:1 relationship between machine and code. If that were the case, his code would have been always understood, always self-evident, never in need of revision, and always independent of his historical and situational matters. If such plainness were evident, Ray's slicing would not have necessitated his ensembled measures to understand the code and data, his various historical and contextual senses of the two textual forms, and the developing goals of the project. By applying this intermediary writing framework, this study expands what counts as writing and how to define code and data as objects of study. Future research can examine ensembles, spectrum of durabilities, and senses to theorize how code and data—and writing in all its forms—are interconnected across myriad senses and communicative media, rather than discretely separated from them. By drawing these intermediations together, writing studies can better explain how and why writers perceive, construct, and negotiate rhetorical factors through their writing.

Appendix A

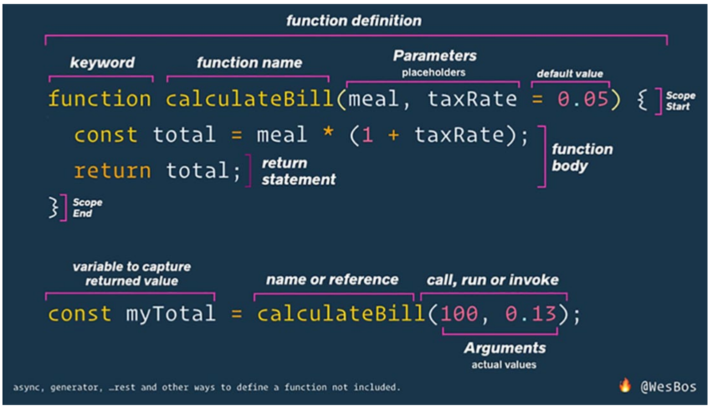
Reading Basics of JavaScript, Data Structures, and File Formats

> *[Ray watches himself review the data set some more in Calc.]*
> *Ray: Again, just trying to determine which fields [in the data set] are multiple fields—have multiple values. So, I'm looking at the data to visually see it and then coding it to an array, essentially, if it needs to be. (emphasis added, Retrospective Account)*

The above excerpt embodies an important element of Ray's slicing activity: coding to look and see how his code was writing with data. It turns out that this focus on seeing, an unfortunate abled-body trope, is commonly shared across coding domains. Prominent practitioner Victor (2013) defined computer coding as the act of "blindly manipulating symbols" (see 8 minutes, 21 seconds). Through such a definition, Victor emphasizes how coding involves the persistent editing and rerunning of a program, so a person can "see" and review any potential changes to the code's output. This gap between coding and knowing what the code will do is a central experience of anyone who codes, regardless of one's goals. In this case of Ray, he writes code in JS that takes input data in some format and/or structure, then his coding rewrites that data.

For readerly support, I describe some basic features of the JS programming language and two main data formats for people with little to no coding experience. Akin to any written language, programming languages include recognizable scopes of microlevel acts of writing that guide the programmer and define the computer's interpretation of the code. For example, in JS, curly braces **{. . .}**, square brackets **[. . .]**, and parentheses **(. . .)** define different forms of computational *scope*. Without getting mired in the details, these grammatical marks signal the opening and closing of a variety of operations. Other marks, such as the semicolon (;) in JS, demarcate the completion of statements and expressions that perform an operation on data.

In this context, Ray wrote JS in what is called a *functional* style. Functional style defines a series of computations on the input data to transform said data as an output. In Figure 7, I offer an annotated example of a JS function to help any person who may be unfamiliar with JS code and its functions (Bos, 2019). In this example, it is important to know that data variables are *instantiated* into a computer's memory with the keyword **var**:



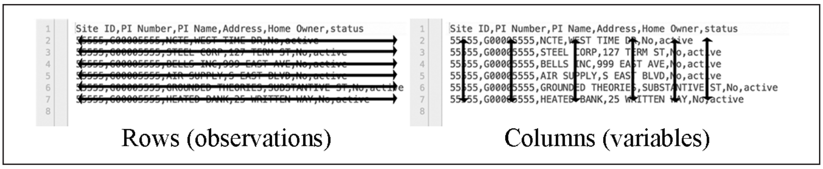
```
function calculateBill(meal, taxRate = 0.05) {
  const total = meal * (1 + taxRate);
  return total;
}

const myTotal = calculateBill(100, 0.13);
```

Figure 7. Labeled example of a JavaScript function (Bos, 2019), which enables a coder to call it, send it data arguments, transform the data through a specified computational means, and return it back to the place the function was called.

var myTotal. The value computed by the **calculateBill()** function, which includes “arguments” or values to be used by the function, is returned to this precise location. The returned value is “assigned” to the variable name, **myTotal**. Assignments occur in a right-to-left position. Multiple functions can be written to perform on data in a single file or across a broader system of files. Accordingly, digital data are transformed numerous times to arrive at the explicit output aim for the task, which is the central aim of this case-study.

Different programming languages share lineages in how data can be rewritten and rerendered across multiple forms, structures, file formats for creative and consequential ends. Most of the data sets that Ray inputs into and changes with his code have become standardized via digital *formats*. In other words, computers and high-level programming languages have been encoded by standards-bodies to assume a particular structure for representing these digital data as structured lists of information.

In an interview, Ray noted how he prefers CSV files, but he often receives Microsoft Excel spreadsheet files (.xls) or PDF files from external sources, such as government entities. CSV have long been used to convert and exchange spreadsheet file data across numerous other file formats. As the CSV name suggests, it assumes that the information will be separated by commas, but it also includes other parsing rule operations that are heavily influenced by two-dimensional tabulated data sets. The CSV file format



```
Site ID,PI Number,PI Name,Address,Home Owner,status
55555,G00005555,NCTE,WEST TIME DR,No,active
55555,G00005555,STEEL CORP,127 TERM ST,No,active
55555,G00005555,BELLS INC,999 EAST AVE,No,active
55555,G00005555,AIR SUPPLY,S EAST BLVD,No,active
55555,G00005555,GROUNDED THEORIES,SUBSTANTIVE ST,No,active
55555,G00005555,HEATED BANK,25 WRITTEN WAY,No,active
```

```
Site ID,PI Number,PI Name,Address,Home Owner,status
55555,G00005555,NCTE,WEST TIME DR,No,active
55555,G00005555,STEEL CORP,127 TERM ST,No,active
55555,G00005555,BELLS INC,999 EAST AVE,No,active
55555,G00005555,AIR SUPPLY,S EAST BLVD,No,active
55555,G00005555,GROUNDED THEORIES,SUBSTANTIVE ST,No,active
55555,G00005555,HEATED BANK,25 WRITTEN WAY,No,active
```

Figure 8. Example comma-separated values with an overlay of arrows, which display its "flat," two-dimensions: horizontal rows and vertical columns. Rows are considered the indexed observations, except the highlighted first row displays the "header" row, which defines the variables as columns.

assumes that people learn and understand that the list has two-dimensions, rows and columns, which follow from tabulated data: rows as the thing being observed, and columns as the variable properties of that thing. Additionally, the first line of the CSV file is defined as the header, which names each column that denotes the potential variables for use. Overall, CSVs translate spreadsheets for the digital medium, where best practices expect observations to be placed along rows and variables about those observations within each "cell" of their respective column within this two-dimensional table (see Figure 8).

JSON is named as such, since its development occurred with the JS language and its importance to coding languages for the web.[17] Regardless of this language connection, JSON was developed as an "interchangeable format," which simply means that it was designed to work across all major languages; notably in the linguistic family of the C language.[18] JSON data are *objects* that store an unordered set of keyed name-value pairs. An *object* is a particular type of orientation to representing a thing or concept. In contrast to a CSV, which represents phenomena with two dimensions, a JSON object can spatially and syntactically represent hierarchical relationships of a noun's properties. It does this through establishing an assumed unordered set of key-value pairings of variable data and their formats, which can nest more objects with their own key-value pairs.

According to the official JSON (n.d.) introductory page, the basic object syntax is to denote how an "object begins with {*left brace* and ends with }*right brace*. Each [keyed] name is followed by :*colon* and the name/value pairs are separated by [a],*comma*" (para. 6). For example, a person can be an object with certain properties, based on the goals, contexts, and needs of the people collecting and using the data. A person, as a JSON object in Ray's HT campaign, might be represented with the following properties, where ellipses denote additional encapsulated data between the data-type syntax:

```
{
"phone": "555-555-5555",
"name": "Chris Lindgren",
"state": "VA",
"city": "Blacksburg",
"zip": "24060",
"goal": "news",
"referrer": "email",
"subscribed": True,
"confirmed": True,
"timezone": "America/New_York",
"hourOffset": 0,
"signup": 1,
"received": [{. . .}, . . .],
"sent": [{. . .}, . . .]
},
. . .
{
  . . .
};
```

In this example, based on the code from Ray's original web-texting application, each person's phone number indexes them with some other identifying information about the person. Each person also has their own trace of both received and sent messages, where received are participant responses to sent messages from the application. For an example railroad diagram, which shows how the JSON object notation has been designed with general parsing rules to be observed by languages, see Figure 9.

Overall, these operations and properties of programming languages and file formats mediate and are mediated by the coders who write with them. After this quick survey of these standardized features, I wish to emphasize how they are not ahistorical and decontextual in their invention and revision. Instead, they are subject to a variety and social-technical factors in perpetual motion, as is every language and their modalities, as people adapt them to community needs, goals, values, insights and biases.

Glossary of Key Terms

The below resource is an abridged list of terms used in this article. If needed, I recommend Mozilla Developer Network's (2020) more comprehensive references about JS.

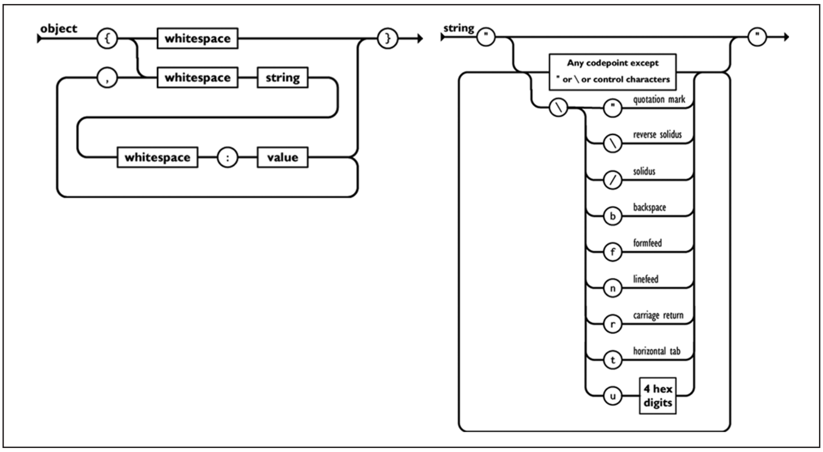


Figure 9. Railroad diagrams of parsing rules and elements for two parts of the JavaScript Object Notation (JSON): Main object's scope (left) and String data type (right) (*JSON*, n/a). Numerous other data types can be used instead of a String, such as array lists, integers, float numbers, and even another object nested as a "child" to its "parent."

Array: A sequential list, starting at zero, of data objects of the same type, whether strings, integers, floats, and so on: **["writing", "a", "list", "of", "strings"]** or **[1, 2, 3]**

Assignment: The assignment operator (**=**) assigns a value to a variable: **var coding = "writing";**

Boolean value: A data type that is assigned to a variable as either **true** or **false**.

Codebase: A large, organized set of source code files that render a software system.

Code library: A collection of modular computational methods that coders can import into their project as a dependency, so they need not write their own version of a similar method over and over again.

Comma-separated values (CSV): A file format that translates spreadsheets for the digital medium, where best practices expect observations to be placed along rows and variables about those observations within each "cell" of their respective column within this two-dimensional table.

Compiler: A codebase that reads in the source code written by the developer and translates the entire source submitted into machine code, as opposed to interpreters.

console.log(): A built-in method in JS that "prints out" data to the console: **console.log("Hello, World!");** would print out the String **Hello, World!**

Data format (also File format): A general term that denotes how data files follow standardized grammar and syntax rules. See CSV or JSON.

Data structure: A general term to describe the numerous data types that programming languages have been developed to read, write, store, and use.

Expression: Any unit of code that resolves to a value:

var myDiscipline

var myDiscipline = "Writing" + "Studies";

Float: A data type of floating-point numbers: 1.1, 3.125, 50.0005, and so on.

Instantiation: The act to create a new variable placeholder: **var writing;**

Integer: A data type of whole numbers: 1, 3, 50, and so on.

Interpreter: A codebase that reads in the source code written by the developer and translates it line-by-line into machine code, as opposed to compilers.

JavaScript: Many high-level programming languages' source code are compiled into computer byte code. JS is contentiously referred to as an interpreted language, which differs in that it is not compiled at run-time. For a more comprehensive description of the JS compiling and interpreters, read Simpson's (2014) *You Don't Know JavaScript* book series, and specifically *Scopes and Closures*. It can be read for free on his Github repository of the book series: https://github.com/getify/You-Dont-Know-JS.

JavaScript Object Notation (JSON): A file format and data object that stores an unordered set of keyed name-value pairs: {"intermediary": ["text sense", "ensemble", "spectrum of durability"]}. An object is a particular type of orientation to representing a thing or concept.

Map object: A data object with key-value pairs and retains the original insertion order of its keys:

var tweets = new *Map*()

tweets.*set*('1000293845', {tweet: "Hi, everyone. #teamrhetoric", hashtags: ["teamrhetoric"]})

Node.js: According to nodejs.org (2020), Node.js is an "asynchronous event-driven JS runtime built on Chrome's V8 JavaScript engine." It is technically compiled and interpreted, so it can be run in a terminal and can also handle server requests, which was not an original feature of the JS language. In sum, Node.js can handle concurrent connections and mitigates memory issues with a garbage collector, that is, digital memory not being used, which JS does not implement. These are more technical matters that warrant their own examinations.

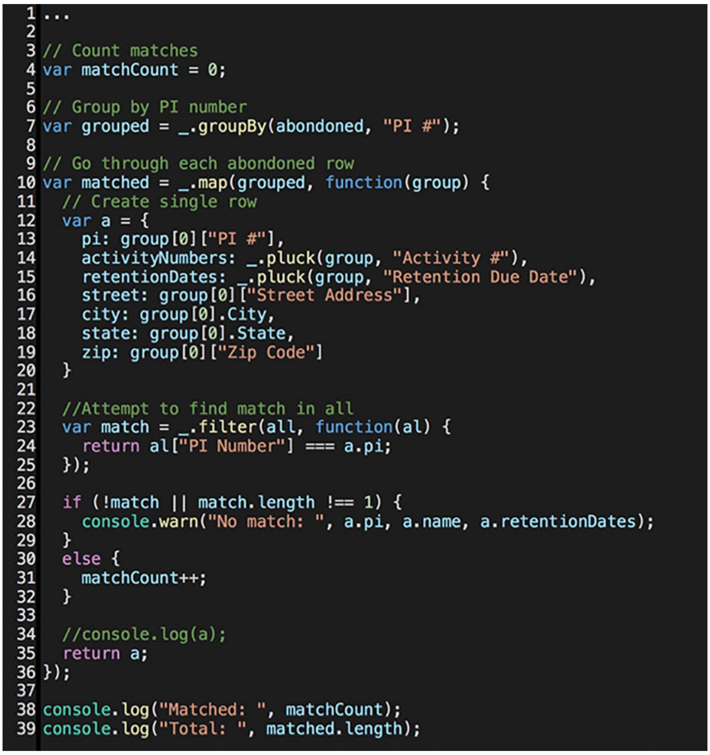
Operator: A grammatical mark that represents multiple types of operations: **=** and **+=** are example form assignment operators; **&&** and **||** are binary logical operators; and so on.

String: A data type that represents any sequence of characters encapsulated by quotation marks: "Chris Lindgren."

Variable: A named value that help contextualize any JS data type: **var myName = "Chris Lindgren";**

Appendix B

More Complete Code Excerpt From the Toxic Sites Slicing Task



```
...

// Count matches
var matchCount = 0;

// Group by PI number
var grouped = _.groupBy(abondoned, "PI #");

// Go through each abondoned row
var matched = _.map(grouped, function(group) {
  // Create single row
  var a = {
    pi: group[0]["PI #"],
    activityNumbers: _.pluck(group, "Activity #"),
    retentionDates: _.pluck(group, "Retention Due Date"),
    street: group[0]["Street Address"],
    city: group[0].City,
    state: group[0].State,
    zip: group[0]["Zip Code"]
  }

  //Attempt to find match in all
  var match = _.filter(all, function(al) {
    return al["PI Number"] === a.pi;
  });

  if (!match || match.length !== 1) {
    console.warn("No match: ", a.pi, a.name, a.retentionDates);
  }
  else {
    matchCount++;
  }

  //console.log(a);
  return a;
});

console.log("Matched: ", matchCount);
console.log("Total: ", matched.length);
```

Figure 10. Extended excerpt from Ray's matching task, during the toxic sites project. The ellipses denote excluded code. Note how this is an excerpt during the task, so it only represents that particular moment—not what the code came to be as a persistent slice within the final project.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was supported by the University of Minnesota's Graduate Research Partners Program Fellowship from the College of Liberal Arts, Affiliates 21st Century Summer Research Fellowship in Professional and Technical Communication, James L. Brown Semester Fellowship in Rhetoric, and their Medtronic/Industrial Affiliates Program Fellowship.

Notes

1. All names, places, and artifacts have been altered to maintain confidentiality of participants in this institutional review board–approved case study (University of Minnesota No. 1509P78181 & Virginia Tech No. 17-924). This quote by Ray is from semistructured interview No. 1.
2. Job titles for people who program are many—contentious even. In this article, I use *developer* and *coder* interchangeably, since "Frontend Developer" was Ray's official job title. However, I recognize that some titles signal different domains and levels of experience. For more information about data journalism, see Seth C. Lewis's (2015) edited special issue in Digital Journalism on "Journalism in an Era of Big Data: Cases, Concepts, and Critiques."
3. At this moment, computer coding remains a relatively unfamiliar practice to most writing researchers. Considering this gap in experience, I have provided a basic coding guide specific to Ray's context, which describes the JS programming language and its data structures and formats in Appendix A. It can be read either before engaging this article, or it can be used as an as-needed resource. From this point forward, I assume that the reader has a basic sense about how to read JS functions and a few data formats (CSV and JSON) and data types (strings, integers, arrays, etc.) to engage the excerpted materials.
4. See also Roundtree's (2013) analysis of how scientists developed computer simulations through data deliberations.
5. See Essinger (2007), Nofre et al. (2014), Rojas (2002), and Wexelblat (1981) for historical treatments of computer programming languages and coding practices prior to programming languages. In brief, it involved heaps of punch cards, sometimes special pseudocode paper for planning how to punch the cards, and numerous tasks coordinated across people, tools, and machines to facilitate the process.
6. See Byrd (2019) on racialized logics operating in a coding bootcamp as African American learners seek to develop coding literacies, and Easter (2020) on gendered logics operating in esoteric programming languages.

7. Bracewell and Witte developed *work ensembles* from Vygotsky's mediation and cultural-historical activity theory, after reviewing the limitations of activity theory at the time. I do not take up activity theory, but it may offer alternative frameworks to study coding as writing.
8. It is worth noting one key difference between Vygotsky's conception of mediation from Hutchins's: Vygotsky theorized conceptual development in terms of reconciliation (cf. White, 2014): that a person can control their environment to develop their consciousness. Hutchins (1995, pp. 283-285) argued that reconciliation is not always a consequence of cognition, since people establish interactions—intermediaries—among their mediating ensemble to complete a task.
9. In Hutchins's case, he observed a "computational ritual" of the ship navigation crew's fix-cycle. He tested whether or not the crew had developed a "positional consciousness" (p. 26) in relationship with the mediating structure of the navigation deck—an insight and research problem not far removed from text sense. An example of this sense included how an expert quartermaster chief could immediately sense that a novice plotter's bearing coordinates was inaccurate on hearing it (p. 141).
10. Multiple participants used "slice" and "slicing" at some point (Ray, Vince, and Phil), but practitioners across data-driven domains used these terms too. For example, Microsoft Excel has even produced a feature called "slicers" to denote the act to select and filter pivot tables.
11. This number captures merely what I observed. Additionally, there are other types of work Ray fulfilled that was difficult and/or not as integral to inscribe, such as conducting a quick code review for a team member or helping someone think through a one-off problem about code or data set.
12. The team also used Github, which is a version control system to help the team share and develop their projects remotely. While important to the team, it did not become a focus for this particular case.
13. Ray's JS is in the runtime Node.js family standard. See the glossary for more information.
14. In this situation, Ray imported the *Lodash* library (Sirois & Hall, 2019). According to Lodash's developers Sirois and Hall (2019) in the "Why Lodash?" section, it "makes JS easier by taking the hassle out of working with arrays, numbers, objects, strings, etc." (para. 1).
15. MySQL is a Structured Query Language to design relational databases. It is a language designed to help structure, add, search, filter, remove, and so on large amounts of data.
16. See Rawson and Muñoz (2019) and Au (2020) for a review of concerns surrounding the limited critical engagement with processing data.
17. See Douglas Crockford's (2011) personal history on the development of JSON standards.
18. See Lévénez's (n.d.) upkeep of a programming language lineage chart, which was once featured by O'Reilly Media.

References

Au, R. (2020, September 5). Data cleaning IS analysis, not grunt work. *Counting Stuff*. https://counting.substack.com/p/data-cleaning-is-analysis-not-grunt

Bazerman, C., Applebee, A. N., Berninger, V., Brandt, D., Graham, S., Matsuda, P. K., Murphy, S., Rowe, D. W., & Schleppegrell, M. (2017). Taking the long view on writing development. *Research in the Teaching of English*, *51*(3), 351-360.

Benjamin, R. (2019). *Race after technology: Abolitionist tools for the new Jim Code*. Polity.

Bizzell, P. (2003). Cognition, convention, and certainty. In V. Villanueva (Ed.), *Cross-talk in composition theory: A reader* (pp. 387-411). National Council of Teachers of English.

Bos, W. [@wesbos]. (2019, March 14). *JavaScript functions visualized* [Image attached] [Tweet]. Twitter. https://twitter.com/wesbos/status/1105907924088565762

Bracewell, R. J., & Witte, S. P. (2003). Tasks, ensembles, and activity: Linkages between text production and situation of use in the workplace. *Written Communication*, *20*(4), 511-559. https://doi.org/10.1177/0741088303260691

Brown, B. (2006). "The next line": Understanding programmers' work. *TeamEthno-Online*, (2), 25-33.

Bucher, T. (2018). *If . . . then: Algorithmic power and politics*. Oxford University Press.

Byrd, A. (2019). Between learning and opportunity: A study of African American coders' networks of support. *Literacy in Composition Studies*, *7*(2), 31-55. https://doi.org/10.21623/1.7.2.3

Chun, W. H. K. (2005). On software, or the persistence of visual knowledge. *Grey Room*, (18), 26-51. https://doi.org/10.1162/1526381043320741

Chun, W. H. K. (2011). *Programmed visions: Software and memory*. MIT Press. https://doi.org/10.7551/mitpress/9780262015424.001.0001

Clayson, A. (2018). Distributed cognition and embodiment in text planning: A situated study of collaborative writing in the workplace. *Written Communication*, *35*(2), 155-181. https://doi.org/10.1177/0741088317753348

Crockford, D. (2011, August 28). *Douglas Crockford: The JSON Saga* [YouTube Video]. YouTube. https://www.youtube.com/watch?v=-C-JoyNuQJs

Doheny-Farina, S. (1993). Research as rhetoric: Confronting the methodological and ethical problems of research on writing in nonacademic settings. In R. Spilka (Ed.), *Writing in the workplace: New research perspectives* (pp. 253-267). Southern Illinois University Press.

Dyson, A. H., & Genishi, C. (2005). *On the case: Approaches to language and literacy research*. Teachers College Press.

Easter, B. (2020). Fully human, fully machine: Rhetorics of digital disembodiment in programming. *Rhetoric Review*, *39*(2), 202-215. https://doi.org/10.1080/07350198.2020.1727096

Essinger, J. (2007). *Jacquard's web: How a hand-loom led to the birth of the information age*. Oxford University Press.

Farkas, K., & Haas, C. (2012). A grounded theory approach for studying writing and literacy. In K. Powell & P. Takayoshi (Eds.), *Practicing research in writing studies: Reflexive and ethically responsible research* (pp. 81-96). Hampton Press.

Galloway, A. R. (2006). *Gaming: Essays on algorithmic culture*. University of Minnesota Press.

Greene, S., & Higgins, L. (1994). "Once upon a time": The use of retrospective accounts in building theory in composition. In P. Smagorinsky (Ed.), *Speaking about writing: Reflections on research methodology* (pp. 20-54). Sage.

Haas, C. (1996). *Writing technology: Studies on the materiality of literacy*. Lawrence Erlbaum.

Hayles, K. (2005). *My mother was a computer*. University of Chicago Press. https://doi.org/10.7208/chicago/9780226321493.001.0001

Higgins, A. (2007). "Code talk" in soft work. *Ethnography*, *8*(4), 467-484. https://doi.org/10.1177/1466138107083563

Hopper, G. (1978). Keynote address. In *Proceedings of the history of programming languages conference* (pp. 7-19). Association for Computing Machinery. https://doi.org/10.1145/800025.1198341

Hutchins, E. (1995). *Cognition in the wild*. MIT Press. https://doi.org/10.7551/mitpress/1881.001.0001

Imbrenda, J. P. (2016). The blackbird whistling or just after? Vygotsky's tool and sign as an analytic for writing. *Written Communication*, *33*(1), 68-91. https://doi.org/10.1177/0741088315614582

JSON. (n.d.). *Introducing JSON*. http://www.json.org/

Kaggle. (2017). *The state of machine learning and data science* [Report]. https://web.archive.org/web/20181130112939/https://www.kaggle.com/surveys/2017

Katz, S. M. (2002). Ethnographic research. In L. Gurak & M. M. Lay (Eds.), *Research in technical communication* (pp. 23-46). Praeger.

Kittler, F. A. (1997). There is no software. In J. Johnston (Ed.), *Literature media information systems* (pp. 147-155). G & B Arts International.

Kittler, F. A. (1999). *Gramophone, film, typewriter* (G. Winthrop-Young & M. Wutz, Trans.). Stanford University Press.

Knuth, D. E. (1968). *The art of computer programming* (*Vol. 1*). Addison-Wesley.

Ko, A. J. (2016). What is a programming language, really? In *Proceedings of the 7th international workshop on evaluation and usability of programming languages and tools* (pp. 32-33). Association for Computing Machinery. https://doi.org/10.1145/3001878.3001880

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. In *Proceedings of the 29th international conference on software engineering (ICSE'07)* (pp. 344-353). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/ICSE.2007.45

Ko, A. J., LaToza, T. D., & Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, *20*(1), 110-141. https://doi.org/10.1007/s10664-013-9279-3

Lévénez, E. (n.d.). *Computer languages history*. https://www.levenez.com/lang/

Lewis, S. C. (2015). Journalism in an era of big data. *Digital Journalism*, *3*(3), 321-330. https://doi.org/10.1080/21670811.2014.976399

Mackiewicz, J., & Thompson, I. (2014). Instruction, cognitive scaffolding, and motivational scaffolding. *Composition Studies*, *42*(1), 54-78.

Manovich, L. (2002). *The language of new media*. MIT Press.

Mei, S. (2014, July 15). Programming is not math. *Sarah Mei*. http://www.sarahmei.com/blog/2014/07/15/programming-is-not-math/

Mozilla Developer Network. (2020, August 16). *JavaScript: Reference*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript

Noble, S. U. (2018). *Algorithms of oppression: How search engines reinforce racism*. New York University Press. https://doi.org/10.2307/j.ctt1pwt9w5

Nofre, D., Priestley, M., & Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950-1960. *Technology and Culture*, *55*(1), 40-75. https://doi.org/10.1353/tech.2014.0031

Opel, D. S., & Hart-Davidson, W. (2019). The primary care clinic as writing space. *Written Communication*, *36*(3), 348-378. https://doi.org/10.1177/0741088319839968

Pigg, S. (2014). Emplacing mobile composing habits: A study of academic writing in networked social spaces. *College Composition and Communication*, *66*(2), 250-275.

Prior, J., Robertson, T., & Leaney, J. (2006). Programming infrastructure and code production. *TeamEthno-Online*, (2), 112-120.

Rawson, K., & Muñoz, T. (2019). Against cleaning. In M. K. Gold & L. F. Klein (Eds.), *Debates in the digital humanities*. University of Minnesota Press. https://doi.org/10.5749/j.ctvg251hk.26

Rojas, R. (Ed.). (2002). *The first computers: History and architectures*. MIT Press.

Roundtree, A. K. (2013). *Computer simulation, rhetoric, and the scientific imagination*. Lexington.

Sano-Franchini, J. (2018). Designing outrage, programming discord: A critical interface analysis of Facebook as a campaign technology. *Technical Communication*, *65*(4), 387-410.

Schriver, K. (1991). *Plain language for expert or lay audiences: Designing text using protocol-aided revision* (Technical Report No. 46). University of California, Berkeley & Carnegie Mellon University. https://archive.nwp.org/cs/public/download/nwp_file/85/TR46.pdf?x-r=pcfile_d

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434-451. https://doi.org/10.1109/TSE.2008.26

Simpson, K. (2014). *You don't know JavaScript: Scope and closures*. O'Reilly.

Sirois, J.-P., & Hall, Z. (2015). *Lodash documentation* (Version 3.10.1) [Documentation]. Lodash. https://lodash.com/docs/3.10.1#groupBy

Sirois, J.-P., & Hall, Z. (2019). Lodash [Overview]. Lodash. https://lodash.com

Smagorinsky, P. (2001). If meaning is constructed, what is it made from? Toward a cultural theory of reading. *Review of Educational Research*, *71*(1), 133-169. https://doi.org/10.3102/00346543071001133

Spradley, J. P. (1979). *Ethnographic interview*. Wadsworth, Cengage Learning.

Swarts, H., Flower, L., & Hayes, J. (1984). Designing protocol studies of the writing process: An introduction. In R. Beach & L. S. Bridwell (Eds.), *New directions in composition research: Perspectives in writing research* (pp. 53-71). Guilford Press.

Thompson, I. (2009). Scaffolding in the writing center: A microanalysis of an experienced tutor's verbal and nonverbal tutoring strategies. *Written Communication*, *26*(4), 417-453. https://doi.org/10.1177/0741088309342364

Vee, A. (2017). *Coding literacy: How computer programming is changing writing*. MIT Press. https://doi.org/10.7551/mitpress/10655.001.0001

Victor, B. (2013, May 13). *Drawing dynamic visualizations* [Video]. Stanford HCI seminar. Vimeo. https://vimeo.com/66085662

Vygotsky, L. (1987). Thinking and speech. In R. Rieber & A. Carton (Eds.), *L. S. Vygotsky, collected works* (N. Minick, Trans.; *Vol. 1*, pp. 39-285). Plenum Press. (Original work published 1934)

Wexelblat, R. L. (Ed.). (1981). *History of programming languages*. Academic Press.

White, E. J. (2014). Bakhtinian dialogic and Vygotskian dialectic: Compatibilities and contradictions in the classroom? *Educational Philosophy and Theory*, *46*(3), 220-236. https://doi.org/10.1111/j.1469-5812.2011.00814.x

Wickman, C. (2010). Writing material in chemical physics research: The laboratory notebook as locus of technical and textual integration. *Written Communication*, *27*(3), 259-292. https://doi.org/10.1177/0741088310371777

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition*. Addison-Wesley.

Witte, S. P. (1992). Context, text, intertext: Toward a constructivist semiotic of writing. *Written Communication*, *9*(2), 237-308. https://doi.org/10.1177/0741088392009002003

Yin, R. K. (2014). *Case study research and applications*. Sage.

Author Biography

Chris A. Lindgren is an Assistant Professor in the Department of English at Virginia Tech. His research areas include literacy studies, digital cultural rhetoric, and the rhetorics of data. He teaches in the undergraduate Professional and Technical Writing program and graduate Rhetoric and Writing program.