Design and Preliminary Evaluation of MIDST, a System to Support Stigmergic Coordination in Data-Science Teams

KEVIN CROWSTON, JEFFREY S. SALTZ, AMIRA REZGUI, and YATISH HEGDE, Syracuse University
SANGSEOK YOU, HEC Paris, France

We describe the design, implementation and preliminary evaluation of MIDST, a system to support stigmergic coordination in data-science teams. We first define a theoretical model of stigmergic coordination, that is, coordination supported by a shared work product. We hypothesize that stigmergic coordination depends on three socio-technical affordances, the visibility and combinability of work, along with defined genres of work contributions. We describe the implementation of a system, MIDST, that supports these affordances and that we expect to support stigmergic coordination. We conclude with an initial assessment of the impact of the tool on the work of project teams of three to six data-science students. Our initial findings suggest that even using an early version of the system, MIDST users perceived improved workload fairness and fewer team output coordination issues, while spending less time on explicit coordination, suggesting that the system was in fact useful in supporting stigmergic coordination, supporting our hypotheses.

CCS Concepts: • Human-centered computing → Computer supported cooperative work; Asynchronous editors: Empirical studies in collaborative and social computing; • Information systems → Data analytics.

Additional Key Words and Phrases: stigmergic coordination, translucency, awareness, data-science teams

ACM Reference Format:

1 INTRODUCTION

Data science is an emerging discipline that combines expertise across a range of domains, including software development, data management and statistics. Data-science projects typically have a goal of identifying correlations and causal relationships, classifying and predicting events, identifying patterns and anomalies and inferring probabilities, interest and sentiment [25]. A common data-science tool is R [1]: analyses are performed by writing what are essentially programs in the R language that take data as input and output analysis results. While small analyses can be performed by an individual, larger projects require teams of analysts working together.

Much has been written about the development of new data-science algorithms that can be used to generate useful insights. Unfortunately, less has been written about other challenges that might be encountered when working as a data scientist [57]. Data-science projects need to focus not
only on algorithms but also on people, process and technology [30, 31]. The need is recognized for more guidance on how data scientists can work together. In particular, the main challenge in a data-science project is task coordination [29].

The goal of the project described in this paper is to better support coordination in data-science teams by transferring findings about coordination from another setting, namely free/libre open source software (FLOSS) development, to the setting of data-science teams. We do so by: 1) identifying a novel form of coordination that appears to be part of the success of FLOSS development, namely stigmergic coordination [9], 2) theorizing socio-technical affordances that support stigmergic coordination in FLOSS and considering how these might be applied in a data-science setting, and 3) developing a system that implements these affordances and which we therefore hypothesize will support stigmergic coordination. We conclude with 4) a preliminary evaluation of the system. The work thus provides not only a useful system but also a test of our theory of stigmergic coordination. However, we emphasize that the evaluation is just preliminary and that the main contribution of the paper is a theory-driven system design and implementation (i.e., steps 2 and 3).

1.1 Stigmergic coordination

In this section, we draw on prior work on FLOSS coordination to describe the paradox that motivates the project: the apparent ability of distributed teams to coordinate with little or no explicit communication. This finding emerged from studies of how FLOSS developers coordinate [10, 37, 38]. Somewhat unexpectedly, these studies found little evidence of overt coordination: FLOSS developers seemed to rarely communicate about coding tasks. The lack of evidence was surprising considering the transparency of FLOSS projects. It was expected to find direct, discursive communication in email or other discussion fora through which developers interact but there were few examples. The lack of direct interaction around the work has echoes in other research findings. For example, research has found that developers mostly self-assign work rather than have it assigned to them [20, 21] and often make decisions about code without explicitly evaluating options [33, 34]. Interestingly, when developers do discuss their work, they often refer directly to the software code.

In light of these findings, researchers have theorized that FLOSS development work can be coordinated at least in part through the code, the outcome of the work itself, a mode of coordination analogous to the biological process of stigmergy [32]. Heylighen defines stigmergy thusly: "A process is stigmergic if the work... done by one agent provides a stimulus (‘stigma’) that entices other agents to continue the job" [35]. Accordingly, stigmergic coordination can be defined as coordination based on signals from the shared work. For example, ants follow scent trails to food found by other ants, thus assigning labour to the most promising sources without the need for explicit interaction. The organization of the collective action emerges from the interaction of the individuals and the evolving environment, rather than from a shared plan or direct interaction.

While stigmergy was formulated to explain the behaviour of social insects following simple behavioural rules, it has also been invoked to explain human behaviours: the formation of trails in a field as people follow paths initially laid down by others (similar to ant trails), or markets, as buyers and sellers interact through price signals [52]. For humans and intelligent systems, the signs and processing can be more sophisticated than for insects [53]. For example, the shared environment can be a complex workspace including annotations. Tummolini & Castelfranchi [68] developed a typology of different kinds of messages possible from signs, such as having the ability to do something, having done something or having a goal. In CSCW, Christensen [16–18] discussed how architects and builders coordinate their tasks through “the material field of work” such as drawings, building on earlier work in CSCW focusing on coordination through the “field of work”, including changes in shared databases [59].
Stigmergy has been suggested in particular as an interpretation of how FLOSS developers coordinate [9], what Kalliamvakou et al. called a “code-centric collaboration” perspective [39]. FLOSS developers mostly work with the code that they are developing, managed with source code control systems such as Git that provide status about the state of the code and development. Stigmergy has also been argued as a mechanism in online work more generally. Elliot [27] argued that “[c]ollaboration in large groups is dependent on stigmergy,” with the specific example of authoring on Wikis.

The question then is how work products can support coordination. From this perspective, we state a more specific question for our theorizing in this paper: What socio-technical affordances of shared-work systems enable stigmergic coordination? By socio-technical affordances, we mean the features of the technology used and the practices around that technology. For example, the source-code control systems commonly used by FLOSS developers provide notifications of code submissions; details of the implementation of this technical feature enable other developers to maintain awareness of the state of the code to support coordination. To interpret these change messages, developers likely need some level of technical skill and mental models of the code structure, another kind of affordance. They may also be accustomed to creating code in a way that is easier for others to interpret. The inherent nature of the coding task itself may create the need for specific kinds of coordination that are particularly amenable to stigmergy. If we can identify the socio-technical affordances found in FLOSS development, we may be able to develop a system to support them, thus enabling stigmergic coordination in another context, meaning that people in that context could also achieve well-coordinated work with less explicit effort.

1.2 Related CSCW concepts

Because the goal of this paper is to present a theory-driven system design, we discuss both related theories (in this section) and related systems (in a later section). Stigmergy is related to other concepts of long-standing interest in the field of computer-supported cooperative work (CSCW). First, there has been a stream of research in CSCW and elsewhere that demonstrates the importance of team member awareness for supporting collaborative work [e.g., 14, 15, 17, 26]. Though they are not identical, there is clearly a close relationship between the two ideas about supporting collaboration. Christensen [17] described actions a person might take to make a co-worker aware of an issue, and so distinguishes awareness from stigmergy, as “stigmergy does not entail making a distinction between the work and extra activities aimed solely at coordinating the work”, such as drawing a co-worker’s attention.

Similarly, in contrast to active awareness (one participant calling for the attention of another), Dourish & Bellotti [26] argued for the importance of passive awareness mechanisms, which can be interpreted as supporting stigmergy. Other researchers have proposed awareness displays that allow a team member to develop an awareness of the actions of other team members. Carroll and colleagues [14, 15] examine how awareness can support development of common ground, community of practice, social capital and human development in team. In this project, we focus more narrowly on how awareness of work supports coordination.

A second related concept is system translucency [28] or transparency [19, 23, 24, 63], meaning visibility of details of organizational processes or functions. Consistent with our analysis of stigmergy, Stuart, et al. [63] analyze transparency as a form of information exchange or communication. They note that technology enables new forms of transparency, e.g., as in GitHub, a software development site [22] that provides real-time updates on what other developers are doing. In other words, transparency is a system feature that might support awareness. Researchers have noted similar problems with awareness and transparency, such as the potential for information overload.
from having to review too much information or that making too much visible may inhibit the willingness to share work [7, 23].

As with stigmergy, system transparency provides information that can influence how people work. Dabbish, et al. [24] note specifically that transparency is helpful for coordination. They list numerous uses of visibility information, such as including dependencies with other projects [23]. They further note that being able to see something means “much less need for routine technical communication” [23], suggesting that transparency is substituting for explicit coordination. Research on visibility and transparency can clearly be quite informative for designing systems to support stigmergic coordination. However, this stream of research has not specifically focused on the socio-technical affordances that enable users to make sense of and to use the provided stigma to support coordination, which is the goal of the current theorizing. For example, given the large number of possible signs available, how do developers decide which to attend to?

A third related concept in the CSCW literature is provenance, i.e., the history of a piece of information. Rather than being explicitly and independently created, provenance of documents is built as the documents are changed, or recorded from interaction as the documents are used, i.e., it is a kind of stigma. Hill et al. [36] and Wexelblat & Maes [69] pointed out that knowing how others have interacted with a piece of information can be informative for future interactions with it. Similarly, knowing the history of a document’s development is important in evaluating and knowing how to use it.

In summary, prior research on stigmergic coordination has noted that the shared work product itself can provide the information necessary for coordination. However, this prior work has not yet addressed the question of what socio-technical affordances of shared work systems enable stigmergic coordination. Research in CSCW on awareness, translucency, transparency and provenance provides suggestions for important features, but does not yet fully answer the question.

2 LITERATURE REVIEW

In this section we build an initial theory of the socio-technical affordances that can support stigmergic coordination, starting with the evidence from FLOSS development and extending to our data-science project context. We suggest that these characteristics of systems for sharing work will support coordination of the work, thus distinguishing a system for stigmergic coordination from systems for explicit coordination on the one hand and systems for simple information sharing on the other.

To theorize what affordances of work support coordination, we turn to the literature on documents and work [51]. Code (the shared work in the case of FLOSS development and data science) is a semiotic product recorded on a perennial substrate that is endowed with specific attributes intended to facilitate specific practices [72], thus making it a kind of document. Code differs from other kinds of documents by serving two audiences, one being a machine, the other programmers. However, we focus on the latter, describing properties of code that allows developers or data scientists to share their work with colleagues, and to read, understand and respond to their intentions.

2.1 Documents enabling coordination

Scholars have described how documentation and other accounts of work play a central role in the coordination of work [11, 12, 48–50, 61, 66, 67]. These perspectives have long pointed to the double role of documents as both ‘models of’ work and ‘models for’ work. For the first, documents provide an account of reality as workers manipulate text and other symbolic structures so as to parallel them with reality. For example, data scientists may carefully document the code they have constructed to create a report of the work (analysis) done. This view of work as a document can
be seen in the emerging concept of data-science notebooks [41], which integrate code, comments about the code and the results / visualizations of the code.

But documents also provide a basis from which people further manipulate the world. For example, data-science reports are not simply accounts of work completed: the report, no matter how documented, can also guide ongoing work by suggesting what is left to be done, such as suggesting an attribute requiring further analysis. Taking inspiration from Smith [61] and Bakhtin [4], we suggest that a work product is rarely completely original; it is always an answer (i.e., a response) to work that precedes it, and is therefore always conditioned by, and in turn qualifies, the prior work. What the data scientist does when facing work is responsive and partially determined by what has been going on up until now. The analytical reports are thus accounts ‘for reality’, as they provide a blueprint of the analysis taking shape.

While typically used for exploratory data analysis, the previously-mentioned notebooks provide a hint at treating the data-science analysis as a document, in that these documents provide both ‘models of’ work done and ‘models for’ work to be done. Documents in this way offer a double accountability: when documenting the analysis of a dataset, data scientists mold the account to the reality of the code on their computers and at the same time, mold their ongoing coding to the desires of the client.

Our focus on stigmergic coordination is how documents can serve as a model for work. With this focus in mind, as shown in Figure 1, three further concepts from document studies stand out as helpful in articulating how documents can serve as a model for work: genre, combinability, and visibility and mobility. We address each of these in turn.

![Fig. 1. Theoretical Model of Stigmergic Coordination](image)

### 2.2 Genre and genre systems

A genre is defined as typified action invoked in response to a recurrent situation [70]. People can recognize a document as a model for possible action only because they have some background knowledge about the genre of that document, and thus the expectations associated with that type of communication [48]. For example, common document genres relating to an academic paper include paper submission, reviews, editor’s report, decision letter, reply to reviews, revision, acceptance letter, final submission, galley proof, copyright release and published paper. Each has a characteristic form (e.g., a review template) and purpose. People engage genres to accomplish social actions in particular situations, which are characterized by a particular purpose, content, form, time, place and set of participants.
The same is true of FLOSS work products. A FLOSS developer engages in typified actions invoked in response to recurrent situations. They do so to accomplish a task characterized by a purpose, material form, place, time and participants. By completing a piece of code for colleagues to work on, a developer invokes a specific genre of work. Colleagues will be able to pick up and work with the code (i.e., be able to coordinate their own work stigmergically) because it invokes that genre and so comes with certain expectations. The first engineer might have created a scaffold of a module that simply outlines a structure. In so doing, her work product becomes a model for work associated with specific elements and course of action. It might invoke a sequence of steps or routes to a conclusion. It might invoke certain categories or socio-material arrangements that must be used. In this way, a piece of completed work serves as a model for future work by drawing on its own genre, i.e., what are the expected outcomes, what materials and forms should be invoked at what places and times and by what types of participants.

We expect to also find genres of work in data science, though perhaps not as well defined, given the emerging nature of the field. By completing a piece of analysis (i.e., module or code segment that cleans the data or does one specific analysis), a data scientist invokes a specific genre of work. A colleague will be able to pick up and work with the analysis (module or code segment) if it invokes a genre and so comes with certain expectations. For example, a data engineer might have created a scaffold of a module that reads a dataset and does some basic cleaning. In so doing, this work product might suggest future work associated with specific elements within the dataset.

A key point in the analysis of work in terms of genres is that for genres to enable documents to function as models for work they must be part of the conventions of practice shared among members of particular communities. Genres are not naturally occurring. They are rather learned as part of membership of such communities: As new participants are socialized into the communities, they gradually acquire a naturalized familiarity with the socio-material arrangements and prominent genres.

Furthermore, documents related to work (and so we argue, the work itself) are often organized into what are called genre systems [47], formalized sequences of documents of particular genres providing more or less standardized methods for recognizing what might be done and what does get done as legitimate work. We alluded above to the genre system around an academic paper: submission, reviews, editor’s report, decision letter, revision, acceptance letter, final submission, galley proof, copyright release and published paper. FLOSS development has its own system, e.g., bug reports, patches, tests and releases.

Within a data-science context, a genre system can be viewed as a standardized flow of work. In fact, there are two different potential workflows. First, one can view the status of a module (document) as a kind of genre. For example, within a Kanban project management context, one can view a module as flowing from “to do” to “in progress” to “validate” and then finally to “done” [2]. These phases can be seen as genres because the type of tasks that are appropriate or necessary for a module changes as one moves, for example, from “in progress” to “validate”.

The second form of genre focuses on the nature of work done by the module. In the FLOSS context, source code itself has a structure in which each component has more-or-less well-defined purposes associated with specific functionalities. There are genres of source code: It collectively has the purpose of providing instructions for the computer, but as well, each module of a program has its own specific purpose and so its own subgenre. For example, some modules may manage the interface, while others deal with interactions among data sources.

Moving to the data-science context, we expect to find a different genre system. For instance, an analysis might include steps to first clean the data, then do exploratory analysis, and then execute several different machine learning algorithms and compare the outputs. Thus, if we treat each task as a code module, each code module provide genre expectations and thus serves as a ‘model for’
work at two levels. First, the data-science process includes a number of distinct and typified actions involved in response to a recurrent situation and expressed in a set of characteristic documents, including gaining access to data sources, code to clean the data, analysis code, and so on. These genres are associated with particular purposes. For example, the purpose of the code cleaning is to create a more usable dataset, whereas exploratory analysis is used to provide information to data scientists about the data. Some code is used nearly exclusively by data scientists (e.g., data cleaning), while other code, such as the data analysis, is often shared between clients and data scientists, at least in terms of the results of the code. By looking at these work outputs, experienced data scientists can determine which tasks might be appropriate to do next.

Furthermore, the code module itself has a structure in which each component has more-or-less well-defined purposes associated with particular functionalities. For example, as previously noted, some modules may clean data, while others focus on different specific predictive analytics. For modules to be useful, clarity of communicative purpose is of critical importance. In other words, it should be clear which components are appropriate to modify or add to the analysis, for example, where additional analyses could be added. Identification of the purpose of a module can be promoted via its interface: the inputs to the module, the module name/description and the outputs from the module.

In a well-structured analysis, the clarity of subgenre, or module, is clear. In other words, the subgenre (i.e., the purpose of a specific module) is recognizable and thus, the module is usable by others as a model for work. This includes having well-defined inputs and outputs. In poorly a structured program or analysis, the purpose of particular module may be hard to determine or, in fact, muddled and unclear. This confusion may not directly affect the functionality of the code, but in these cases, the module does not constitute a genre. Future data scientists cannot tell how to add new functionality, such as an additional analysis, because the current work outcomes do not make it clear how to add that new analysis without recreating the entire analysis.

2.3 Combinability

The second important characteristic of work documents for stigmergic coordination is combinability. For the work to be a model for future work, the work must be combinable and improvable in modular increments [38, 44]. Most work tasks are layered and complex: new work contributions can be adjusted and added to existing outcomes. A piece of code might start out as an incomplete frame, a scaffold on which other parts get added in some organized sequence. Later, new functionality can be added to the existing structure.

Combinability is greatly enhanced by the modularity of the code. The modularity of a solution can be considered as a continuum describing the degree to which the components of a solution can be separated, worked on independently, and recombined [58]. Modularity is a familiar concept in programming and has been cited as a key feature of FLOSS code. With respect to data science, the use of R [1] is an example of one aspect of leveraging modularity to improve combinability. Specifically, the Comprehensive R Archive Network (CRAN) contains thousands of “packages” that can be installed and loaded as needed. These packages enable a team to easily combine modules developed by others, such as using an advanced machine learning module via a function call.

However, another aspect of modularity, task modularity, is concerned with how a data-science team breaks down its activities into modules (“chunks of work”) that can be worked on in parallel, but in a coordinated manner. One important benefit of task modularity is that it helps reduce the need to coordinate details of a team member’s work with other team members [6]. This combinability (modularity) enables, for example, data cleaning functionality to be used by predictive analytic modules, either in parallel with the development of data cleaning, or at some point in the future. In this way, new work contributions can be adjusted and added to existing outcomes via the integration...
of additional modules, or the enhancement of one or more modules. Furthermore, a piece of work, such as data cleaning, might start out as an incomplete frame, a scaffold on which other parts get added in some organized sequence. Later new functionality can be added to the existing structure. In this way, an analysis evolves from version to version. Thus, it is not surprising that it has been shown that leveraging modularity delivers significant benefits within many contexts, such as manufacturing [56] and, perhaps most commonly, software development [64].

We note that another benefit of modules is that it supports complex problem-solving by enabling a team member to focus on smaller challenges, rather than needing to focus on the entire problem [6, 13]. A modular approach enables the team to proceed more quickly and effectively [45]. Furthermore, it has also been noted that modularity brings increased flexibility, a better ability to deal with complexity and the accommodation of uncertainty [71]. Thus, one aspect of enabling a team to work well together is by having the team be able to break the project into modular components [6].

Modularity is thus a key to combinability, leveraging a general set of design principles that involves breaking up a problem into discrete chunks [43] and “building a complex product or process from smaller subsystems that can be designed and worked on independently yet function together as a whole” [5]. Hence, the use of modules and genres is likely to be important to data scientists due to the benefits of decomposing tasks and allowing different team members to work on different aspects of the project. In other words, enabling or improving modularity can provide data scientists with an affordance that improves team coordination and effectiveness.

Combinability in FLOSS development is supported by both cultural norms and the source-code control system infrastructure. First, there are strong cultural norms for providing “atomic commits,” that is, developers are encouraged to address only one change or topic when making a commit, leading to many small commits rather than occasional large ones [3]. It is easier to combine code with a focused commit than with a commit that does multiple things and touches bits and pieces of dozens of files in the process. It is likewise easier to back out a focused commit if things should go wrong. Developers are also warned: “Don’t break the build”, which means that the main set of files in the source code control system should always compile and run. This practice ensures that any developer who downloads the code will be able to work with it (i.e., it will be useful for coordinating work), supporting the individual development described above. Combinability in FLOSS development is further supported by the source-code control system infrastructure allowing participants to merge work. For example, they can try out experimental enhancements on the code in a branch before committing it, or work in parallel and then merge their efforts (if their efforts have been modularly defined).

Implementing combinability for data scientists should similarly let them execute and test ideas without interfering with others: they can run the code with their proposed changes and obtain direct feedback about the combinability and thus success or failure of their changes. This approach would allow them to iteratively enhance their understanding of the task and to modify their strategy for managing dependencies between the existing analysis and what they are trying to accomplish. Applied in a data-science setting, data scientists can interact with the code base as they would engage in a conversation by continuously receiving feedback on their output. As a result, data scientists can avoid a lot of communication with other team members, since their active engagement with the artifact (the code) provides substantial insights; one has less need to ask another what their intentions were when one can experiment with the code base.

2.4 Visibility and mobility

The third key feature of documents is their visibility and mobility. Obvious as it may seem, making work visible to others is not a straightforward process. As discussed by Suchman [67], some work may be more visible than other work; some work may cover up previous activity and render it
invisible. For example, service work is notoriously hard to make visible: The better such work is done, the less visible it is to those who benefit from it. CSCW research on awareness and transparency also addresses these concerns. Understanding what elements of work are accessible and how its visibility may change over time is central to understanding how work may or may not serve as a model for future work.

Further, for work to coordinate tasks beyond a physically-restricted space, it must become mobile [44], meaning that it is accessible to others. By being in multiple places, code can coordinate work in multiple settings. Most obviously, FLOSS development infrastructures support the mobility of work by being internet-based. Any FLOSS developer can download the source code from the source-code control system and have access to others’ work as a basis on which they can build their own. As a result, developers can, in many situations, use others’ work as a model for their own work because of their ubiquitous access to the server containing the code. We expect the technical affordance of sharing files to be easy to translate to a data-science setting, though the size of data files may pose a challenge.

Further, many systems provide a mechanism to push changes to other workspaces, rather than having to wait for those others to seek them out. In addition, the source-code control system can also record a revision history: all changes made to each module in the system including what was created or deleted by whom, when. Many changes include short notes that can explain why a change was made (although many changes do not, apparently expecting the reader to examine the code directly). Such histories not only serve as ‘models of’ work but can also point forward by depicting the generally accepted work process. For a newcomer, such histories provide a window to how things are done, what tasks tend to follow what tasks and what is regarded as good and opposed to bad (i.e., reverted) work.

Visibility of FLOSS work is promoted as well through cultural norms about development. A widely-acknowledged culture norm in open source is to “check in early, and check in often.” If people do not share their work often, they are not making it visible to other participants to build on. Large infrequent commits (“code bombs”) increase the chances that there will be conflicts and make it harder for other developers to understand what a change does, again hampering visibility. Indeed, a frequent complaint about a code contribution is that it is too large for developers to easily understand. This cultural practice may be more difficult to translate to data science, as data scientists are not accustomed to thinking in modules. And simply sharing changes made to one large analysis file will not be effective if there are many dependencies among parts of the code, making it difficult to make atomic changes.

2.5 Hypotheses
As shown in Figure 2, we hypothesize that system (specifically, an enhanced interactive development environment or IDE) that implements the affordances discussed above will enable stigmergic coordination. Coordination can be measured by the team’s overall perceived coordination, as well as by a perception of workload fairness (i.e., good division of labor) and fewer reported team output-related issues. Other indications of the effect of the system will be an increase in shared understanding and an impression of support from the system. And because the system supports stigmergic coordination, we hypothesize that the team will achieve these benefits while requiring less explicit coordination effort.

3 SYSTEM DESIGN
Our goal in the project described in this paper is to support stigmergic coordination in data-science teams by transferring findings about coordination to the setting of data-science teams. The previous section laid out three sets of socio-technical affordances that we hypothesized would support
stigmergic coordination: genres, combinability and visibility and mobility. In this section, we discuss how we designed and implemented a data-science team coordination tool to provide these affordances, which we will use to test the hypotheses.

3.1 System Overview

MIDST (Modular Interactive Data Science Tool) is a web-based data-science application that was developed for this project. The tool enables a team of data scientists to collaborate on developing an analysis, which is implemented in the R system. MIDST has three integrated views that team members use to create an analysis (or part of an analysis): the network, task and code views. Each are described below.

3.1.1 Network View. The main view of the analysis is as a workflow, in MIDST’s network view. As with other data-flow tools, the network view helps users break an analysis into smaller chunks of work (nodes), and then visualize the flow of data through the nodes that comprise the analysis. There are three kinds of nodes: executable nodes that contain R code (code modules), data nodes that can be connected to an input of a node and visualization nodes that can be connected to an output. For example, Figure 3 shows a simple analysis that reads in a raw data file (the raw_data.csv node), cleans and saves the data file (the clean.R code node outputting to the clean_data.csv data node) and generate a histogram (the OzoneHist.R code node reading from the clean.R node and outputting to the hist.png visualization node).

In the network view, users can add new nodes, define a node’s inputs and outputs and connect nodes together, implementing a flow of data between the nodes. As users update the network (e.g., adding nodes or connections), the changes are propagated to other users viewing the network. Users can execute the entire network in the network view by pressing the ‘Run’ button at the top of the network view window. Any errors that occur during execution of the network are visible as failed nodes, shown by the exclamation mark in Figure 4. Other controls push or pull code changes or change views (discussed below).

3.1.2 Task View. A second view of the node is a task view, shown in Figure 5. Similar to other task boards, such as Trello (www.trello.com), the status of each code node is indicated by the column it appears in. Users can update the status of a node by simply dragging it to a new column. Tasks (i.e., nodes) can also be created in this view, which will add them to the workflow, but without connections. MIDST’s task view provides a quick overview of the project status: what is being worked on, who is working on it and the overall balance between completed and uncompleted
3.1.3 Code View. Third, by clicking twice on a code node, within either the network or task view, a user drills down to the R code for the node, with the node’s input ports and output ports shown on either side. (Clicking on a data or visualization node gives a preview.) An example is shown in Figure 6, which happens to be the R code for the clean module from Figure 3. An automatically-generated R preamble reads the input ports and makes the data available to the user’s R script as variables with the same name; a postamble takes the contents of the named variables and adds them to the output ports to transfer to other nodes. The author of the R code is responsible for making the connection between these input and output variables.
In this view, similar to how one uses RStudio, the user writes, edits, runs and debugs R code to implement the required functionality for the node. Within the code editor, a user can run the entire node or execute a single line of code (e.g., for debugging). Output from execution is shown in the bottom window pane of the code editor. This output is for the most recent run of the node, whether that was due to the full network being run in the network view, the full node being run in the code editor view, or a specific line being executed within the code editor.

The code in a code node is shared with other users viewing the project. We noted above that the system updates the view of the network dynamically: as users add nodes or connections, these changes are immediately reflected to other users. In contrast, users must explicitly share changes to code, either for one module or the entire network, using a graphical code management system that lets them “push” their updates to and “pull” others’ updates from a centralized code repository. The difference between these two approaches is because as a user edits code, it is likely that their code will often be in an intermediate, non-working condition. If such changes were pushed as they are made, other users would often find that they could no longer run the entire network, which could block their own work (“breaking the build”). Instead, users can wait until their code is in a usable state before sharing it and defer accepting others’ changes until they are ready to incorporate them, since changes could potentially require them update their own code to fit.

Figure 6 shows other collaboration features within the editor. For example, via the status widget on this view, a user can update the status and owner of a node. Team members can post messages about a node in the discussion widget. The MIDST system also posts messages here, e.g., to inform a user when the code being viewed is out-of-date (i.e., that another team member has shared a more recent version of the code).

![Fig. 6. Edit interface for code in a node, showing code, input and output ports and execution output](image)

3.2 Other design decisions
In order to make MIDST user-friendly and easy-to-use, many user-interface design questions had to be addressed. For example, one can hover over an output item and see a preview of that data or easily switch between views. Many of these features are to improve functionality or ease of use.
However, several novel coordination-related requirements emerged as we implemented the system, which we briefly discuss in this section.

3.2.1 System Hints and Notifications. As noted above, users must explicit decide when to share code. To reduce the need for explicit coordination about changes, MIDST proactively suggests when a team member should push an update to the central repository as well as providing reminders when a team member needs to pull the updates from the team’s central repository. MIDST notifies team members, via the discussion widget in the code view or by highlighting the icon in the network view, that another team member has pushed updated code, and also suggests, when viewing the network, that the updated code should be “pulled” from the team’s repository (though as noted above, the timing is up to the user).

MIDST also helps a team member keep the status of their nodes up-to-date. For example, when someone starts to edit a node, if that node is not currently owned by another user, the system suggests that the current user own the node and move it to “in progress”. If a user tries to edit a node owned by another team member, then the system reminds the user that another team member owns that node. Furthermore, when a user pushes code to share it with other team members, MIDST asks if that node’s status should be moved to “validating”.

3.2.2 Shared Execution Environment. Our initial implementation of the system was as an application that would run on the user’s computer to provide the user interface and R session. The application shared network and code changes via GitHub. We soon discovered that sharing code was not enough to enable easy coordination. Users also need to share all the dependencies that the code relies on, such as data files and libraries. Users not infrequently encountered problems running code provided by their teammates, e.g., file paths that needed to be changed or new libraries that needed to be installed. Also, users with less powerful computers faced performance issues.

To avoid these problems, the current version of MIDST provides access to a common computing infrastructure, where each user has a clone of the same computing environment. More specifically, the current implementation is a web application, meaning that all the code runs on a common server. This shared execution environment greatly facilitates team collaboration since issues such as what libraries and what versions are installed as well as other details such as location of data files are eliminated.

3.2.3 Node Ownership. A common problem in a source-code control system is handling multiple changes to the same document, e.g., by merging non-conflicting changes and providing an interface for a person to resolve conflicting changes. We decided to side step this problem. Since a node is assigned to particular a user, in MIDST only the owner of the node is allowed to push changes to the code. If other team members want to make changes, they can change ownership and do it, or discuss the changes with the node owner.

3.3 MIDST’s Support of Stigmergic Coordination

Having reviewed the functionality of the tool, we next discuss how it implements the three sets of socio-technical affordances developed above.

3.3.1 Genres. There are three ways in which MIDST supports genres. First, MIDST supports the notion that each node has a status, which we argue is one form of genre. Second, MIDST support different node types: input nodes, code nodes and output visualization nodes. Finally, there is an implicit set of genres of code nodes in the data-flow pipeline defined in the network view. The network view shows how data flows from, for example, reading data, to cleaning data, to exploratory analysis to more advanced machine learning to outputs. Thus, where the node is within the network provides context as to the type of work being done within that node. Collectively,
these genres helps the data scientist understand the context and goal of the node, independent of the actual details within the node.

3.3.2 Combinability. MIDST supports combinability via the tool’s encouragement for users to create modular components, as well as the ability to merge different users’ work (while also reducing the potential for duplication of work). Specifically, MIDST supports modularity via the network view, where nodes of R code, with clearly defined input and output ports, are connected either to other nodes. These input and output ports act like an interface definition, clarifying the work to be accomplished by the node. With respect to merging work, MIDST supports an easy way to push and pull code, but equally important, a way to note that a node is “owned” by a person on the team. When appropriate MIDST reminds that person to push the code to the central repository and then reminds the other team members to pull it.

3.3.3 Visibility. There are several ways MIDST supports visibility of code status and activity. First, the breakdown of work that is required within the project is clearly shown within both the network and task views. In addition, how the data flows through the system is visualized via the node connections in the network view. Second, MIDST shows node ownership (in the code and task views) and node status. Changes in these node attributes are visually shared in the network view (via the colouring and other node decorations) and in the code view (via messages from a MIDST bot in the discussion widget). Third, nodes with errors are clearly shown in the network view (via the ‘!’ icon on a node). Fourth, changes to code made by other users are highlighted as “nodes that need to be pulled” both via a message in the discussion widget (in the code view) as well as in the network view.

In summary, MIDST offers a shared workplace where the cooperative work is facilitated, not only by requiring active construction from the participants of a common information space, where users can perceive, access, and manipulate the same set of information, but more importantly by providing a shared view. In fact, the core of the notion of a shared view is that multiple actors perceive the same object (the network of node in our case study, containing tasks, description and code or implementation) in the same state and perceive any changes in the state of the object concurrently. Any changes to the object affected by one user will be immediately perceptible to the other users.

3.4 System Implementation

The system architecture for MIDST is shown in Figure 7. The front-end of the system is a web browser. Plotting of the network workflow uses the cytoscape Javascript library. The back-end is a Python web application built using the Flask framework. The web application uses a Mongo database to store data about the network, user projects, code execution and collaboration features and the server file system to store project files, R scripts and data files. Each user has access to his or her projects and the projects that are shared with him or her by the team. The execution of R code in users’ R sessions is managed using the RStudio server software. The instructor has privileges to access all projects of his or her class. The entire system is hosted on an Ubuntu server with 16 cores, 32 GB RAM and 500 GB disk space.

3.5 Comparison to other systems

MIDST has similarities to and draws features from many existing systems, but the purpose and combination of features is unique due to the theory-driven design, making the system a novel contribution.

In some respects, MIDST is an integrated development environment (IDE) for R development, making it comparable to IDE tools such as RStudio. Indeed, a frequent complaint from users is that
its functionality is lacking compared to RStudio. However, RStudio does not support structuring an analysis in modules, which is the goal of the workflow view of MIDST. Nor is it intended to support team collaboration.

A number of workflow tools have been developed to support data analysis, e.g., KNIME. Another research system StatWire [65] shares MIDST’s goals of using a workflow system to promote code modularity. MIDST differs from these systems in two important respects. First, while existing tools are individual tools, MIDST is designed to support a team. Second, the goal of many workflow tools is to enable non-programmers to develop an analysis by combining pre-existing modules. In contrast, in MIDST, we expect users to write their own R code for the code nodes. A more production-oriented version of MIDST could provide a library of nodes, though doing so would break the current one-to-one mapping of code nodes to tasks (discussed below in limitations).

Another feature of MIDST is to provide a way to track and make visible the status of development tasks. There are numerous systems that support task tracking, such as Trello. In contrast to those system, in MIDST the task view is directly tied to the work being done, meaning that a user can click on a task and immediately start editing the code. This connection between a task and a node also helps users to create tasks of an appropriate size and scope. And contrariwise, integration of task statuses is not a feature of other IDE tools.

Another feature of MIDST is code sharing. There are many systems for sharing code and providing updates about the status of the code, such as GitHub. Some of these systems also integrate task tracking. Code sharing may also be integrated with an IDE to make it convenient to check in recently-edited code. However, these tools are designed to facilitate sharing of files. We argued above that these tools do support stigmergic coordination in FLOSS. However, by themselves they are not sufficient to support stigmergy for data science as they do not provide affordances for creating modules that are meaningful for this setting, meaning that the individuals’ work lacks combinability.

A final feature of MIDST is the shared execution environment. R Studio Server similarly allows users to run an R session on a server and data analysis notebooks (e.g., Jupyter [41]) can also provide access to a shared environment. Another approach to managing dependencies is a container system such as Docker [8] that bundles together a program with all of its dependencies. However, these tools do not provide support for modularizing an analysis nor for fine-grained collaboration. For example, Rule et al. [54] observe that notebooks often become difficult to navigate and understand,
which discourages sharing and reuse. To try and address this challenge, they introduced the concept of annotated cell folding (i.e., the ability to hide/unhide blocks of code), which was somewhat helpful, but also caused someone new, who was trying to read the code, to sometimes overlook components of the analysis that were hidden.

Finally, there are many, many collaborative systems designed to support groups, specifically to support coordination of group work (that is, for managing dependencies among group tasks). However, only a few systems have been explicitly aimed at supporting stigmergic coordination. Musil et al. [46] proposed the concept of a Stigmergic Information System (SIS) architecture metamodel, though their goal is to develop an architectural model that describes many kinds of systems rather than to build one. Most systems described as stigmergic appear so far to simply provide access to the shared work, without specific attention to coordination of the work. For example, Zhang et al. [73] described a system for allowing collective construction of a conceptual model.

4 PRELIMINARY EXPERIENCE

We next report our findings from a preliminary evaluation of the system. Specifically, to understand how MIDST supports stigmergic coordination, we report on an exploratory study that compared teams who carried out a data-science project using MIDST to teams that used RStudio. Quantitative and qualitative survey responses were augmented with data from observation of the teams using the two different tools. We emphasize though that the main contribution of this paper is the theory-driven system design and development reported above and this evaluation is just preliminary.

The participants in the study were data-science graduate students. While there has been little written about using students to gain insight into industry teams within the data-science context, experiments with students have been common for decades in the software development domain. In fact, students were used as subjects in 87% of the software development experiments analyzed over a representative ten-year period [60]. It is important to note that when using students as subjects, several factors are typically considered. First, “students vs. professionals” is actually a misrepresentation of the confounding effect of proficiency, and in fact differences in performance are much more important than differences in status [62]. Hence, master-level students, many of whom have several years of industry experience, can often be an appropriate choice for subjects, more so than undergraduate students with minimal experience. Second, comparing across experimental conditions, using students may reduce variability because all students have about the same level of education, leading to better statistical characteristics [42]. Finally, while students might not be as experienced as practicing professionals, they can be viewed as the next generation of professionals and hence many believe they are suitable subjects for studies [40, 55].

Before reporting on these findings about coordination, we note that initial results from other MIDST studies suggest that MIDST improves the modularity of an analysis, as well as the maintainability of an analysis by making it easier to share and to understand the analysis code. Improved modularity and maintainability lead us to believe that MIDST improves combinability (due to improved modular increments) and visibility (since improved maintainability was due in part to improved accessibility and visibility of node inputs and outputs), two of the intermediate variables in our hypothesized model (Figure 1).

4.1 Methodology

There were 231 students, across 11 sections of a graduate-level introductory data-science class. There were nine face-to-face sections and two distance-based sections in the study. Students from
two of the face-to-face sections and one of the distance-based sections used MIDST. There were a total of 72 students in the MIDST condition.

While most of the students in each section were graduate information system students, the course also included students from other graduate programs, mainly business administration or public policy. Each section also had a number of undergraduate students from fields such as information technology, engineering and business.

The students were selected to be in one of the sections of the course, without any knowledge of which sections would use MIDST. The non-MIDST sections used RStudio exclusively and served as a control group. We expected that MIDST would be much more useful for the distance students who have fewer opportunities for interaction and coordination. However, it was very useful in debugging the system to be able to observe and interact with students to understand their problems and to provide support, hence our decision to start with face-to-face sections.

Across all the course sections, students worked on a semester-long group data analysis project, in teams of 3 to 6 people. All students received the same data-science instruction and had similar project requirements. Students who used MIDST were also given instruction on how to use MIDST prior to the start of the project and used it for individual assignments. All students, across all conditions, gained experience using RStudio.

The study protocol was approved by the Syracuse University IRB. Students were not compensated for their participation in the study. Use of MIDST in the MIDST sections was only mandatory for submitting assignments; students could and some did use RStudio for development, as discussed below. Students were informed that MIDST was part of a study and had the option to request that their data not be used; none exercised that option.

4.2 Usage Data

In fall 2018, the class used an earlier version of the tool. The class created a total of 10 team projects. The average team size was 5 students. The average number of nodes per team project was 29.4; the average number of R nodes was 10.6.

In spring 2019, an online class used the current version of the tool. The class created a total of 6 team projects. The average team size was 3.2 students. The average number of nodes per team project was 38.7; the average number of R nodes was 11.8.

4.3 Survey Analysis

To evaluate visibility and overall coordination when using MIDST rather than RStudio, we used a quantitative survey that included scales for Shared Understanding, Workload Fairness, Tool Visibility Facilitation, Perceived Coordination, Coordination Effort, and Team output Issues. The scale items were adapted from other surveys to fit the data science environment. Each item was a 1–5 Likert scale, from disagree (1) to agree (5). See the Scales appendix A for a list of items within each scale, as well as the Cronbach Alpha for each scale (note that all scales had a Cronbach alpha over 0.79). The survey also included open-ended questions that are reported below. The survey was deployed near the end of the class. There were 197 responses to the survey (an 85% response rate). 61 of those students were in the MIDST condition (also an 85% response rate).

4.3.1 Survey Results. Table 1 reports the quantitative survey results. We note that the survey includes students who used an earlier, still somewhat buggy version of the system and only a handful of distance students, those we believe to have a larger need for the system. And even there, not all of the team used MIDST as intended, further compromising the results. Given the preliminary nature of this study, we report differences with a significance level of p<0.15, so these results should be considered as indicative rather than definitive.
A significant result was that there was a perception of less tool support to facilitate collaboration when using MIDST. We attribute this perception to the presence of bugs, UI and performance issues in MIDST, especially in comparison to the polished RStudio application. However, even with that lower level of perceived tool support, team members in the MIDST condition thought there was greater workload fairness. In addition, they perceived fewer team issues relating to task integration (e.g., unusable output or duplicate results). It is also interesting to note that the MIDST teams reported doing less planning (i.e., less coordination effort), while still reporting the same level of shared understanding and perceived coordination.

<table>
<thead>
<tr>
<th></th>
<th>MIDST</th>
<th>NoMIDST</th>
<th>p value of t-test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Understanding</td>
<td>4.3</td>
<td>4.4</td>
<td>–</td>
</tr>
<tr>
<td>Workload Fairness</td>
<td>4.2</td>
<td>4.0</td>
<td>0.14</td>
</tr>
<tr>
<td>Tool Visibility Facilitation</td>
<td>4.1</td>
<td>4.3</td>
<td>0.04</td>
</tr>
<tr>
<td>Perceived Coordination</td>
<td>4.2</td>
<td>4.3</td>
<td>–</td>
</tr>
<tr>
<td>Coordination Effort</td>
<td>4.1</td>
<td>4.3</td>
<td>0.11</td>
</tr>
<tr>
<td>Team Output Issues</td>
<td>2.9</td>
<td>3.1</td>
<td>0.09</td>
</tr>
<tr>
<td>N</td>
<td>61</td>
<td>136</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Survey Results: Average of Responses in the Two Conditions and p value of t-test

4.4 Survey Qualitative Feedback

Two key themes emerged from the student qualitative feedback on the open-ended survey questions. First, student appreciated MIDST’s ability to help them break up the work and stay coordinated within their team. However, they also often commented on bugs, usability and performance issues. One student comment was typical “Though a thoughtful idea, [but] it needs polishing”. Below we list additional student comments for the key themes identified.

MIDST good for sharing/dividing work:
- It is a very good tool for shared projects.
- Task assignment was easy
- MIDST allowed us to divide the work and track the progress of the project.
- We could all work in it simultaneously without asking each other to share the code with us
- Different versions of code can be made available.
- Module wise distribution, easy sharing
- Very good for assigning tasks and group coordination.

MIDST good for visibility/tracking:
- We can track team members work.
- Easily see the progress.
- It can track each member’s progress.

MIDST bugs/issues:
- The basic functions (like ggplot) stop working abruptly.
- Running the code takes a lot of time
- The session timeouts for big networks.
- Created a lot of errors on the codes which were working well on R studio.
MIDST ease of use issues:

- Took us a little time to learn the working
- Identifying and solving errors became extremely challenging for our team.
- The time taken to understand it completely.
- We didn’t have any prior experience working with MIDST, so it took some time in getting familiar with MIDST.

4.5 Finding from Observations

We augment our findings from the survey study with observations of and informal interviews with student teams as they were actually working on the team project. The observations were done by one of the authors sitting in on classes and observing the students working together in small groups, either face-to-face or in a synchronous video conference. The observations were integrated across the two semesters. As appropriate, we note if an example was within the face-to-face or distance-based course. During the observations, the observer would occasionally ask the students to explain what they were doing or what problems they perceived. In reviewing the observations, several themes emerged.

We first note that the project required interaction among the students to build a common understanding of the task. Students in the face-to-face class were able to mobilize all the communicative resources of face-to-face interaction to negotiate a shared understanding of what is to be done. To communicate with each other, students generally meet physically to discuss the main point of their project. During their meeting, students were observed trying to make sure that everyone in the group understands all parts of the project. Students try to help others understand and at the same time feel free to ask for help. Students in the distance section faced more of a challenge, but did have some more limited opportunities for discussion. In both cases, synchronous discussion was augmented with other interaction, e.g., email or a chat group.

In both cases, it was useful to have shared notes to ensure that all members have a common understanding. This was especially important when some members were absent from a discussion. MIDST complemented such unstructured notes by enabling team members to share their understanding of the work to be done, e.g., by giving a meaningful title and a detailed description for each node. As one student said “we note our comments in a shared document or through a group discussion where we describe our fixed goal and the different tasks. However node title and the brief description give me a quick access to the updated goal”.

Beyond easier code sharing, MIDST requires each node to have defined task inputs (which can be the output from others’ node) and output (which can be the input for other’s node). Having well-defined inputs and clear outputs helped team members coordinate their work without discussion. As one student said, “Sometime we don’t ask for resources we just wait for it by looking directly to the needed input from the network.” The tool also provides access to the shared code with its comment, which also supports coordination. As one student said, “I don’t find problems or issues in coordinating the work. I generally, go to the code of each resources and look if it is similar to my expectation. Sometime when the code is clear and well commented, I go directly for modifying it (sometime after permission)”.

The combination of defined inputs and outputs and node ownership also makes explicit coordination easier when it is needed. For example, when students who were waiting for an input from another node’s output finds that the result is different from what was expected, they know who to talk to. Ownership suggests that any modification to the node should be discussed with and validated by the owner. As one said, “If they want to improve one task, they should first ask me and then they go to the code and update it.”
The notification system was also seen as helpful in supporting coordination activities. Providing additional notifications indicating the creation of a new node, the deletion or the update of existing one, the presence of another student in the others node may help tracking others activities specially the related one to their own task. In addition to the frustration expressed about bugs in MIDST, students identified shortcomings. As one said, “I’m not notified about other team member working in the same node and delete it”.

In contrast, students using RStudio lacked a shared virtual workplace, which limited access to information about the project. They used Trello to track tasks and task status. To avoid problems, they tried usually to finish their work on time and post in Trello the status of the work. They thus know when they can ask for the output of another student’s task in case of dependencies. Sharing the code is a challenge for them. Some teams were observed using Google Docs. But since the code is not easily shared, the team members prefer to divide the project in subtasks and work in subgroups of 2 or 3 people, where members in a subgroup can meet frequently and work on tasks that are independent from the other groups. This approach minimizes the need for coordination but potentially increases problems integrating code.

Overall, the majority (but not all) of the teams using MIDST succeeded in modularizing the project and creating the network flow according to their understanding. MIDST seemed easy-to-use as a way to understand, explain and share team members’ project status. All teams agreed that the tool helped them to track project status as the project progressed. A counter-example is provided by one team in the distance course who encountered significant team issues (indeed, the team actually disbanded just before the end of the project). While there were likely many causes for the team’s failure to work well together, one interesting aspect of how they worked was that one of the team members did not do his work within MIDST, but rather used RStudio. When that person tried to integrate their work with the others at the last minute (i.e., a “code dump”), there was significant confusion about what was done and who was doing which analysis. The team members did not understand the analyses done by others on the team. For example, one of the team members used some advanced techniques that the other team members did not know how to interpret (or even if the results were useful). So even though there was significant effort expended by each team member, there was frustration that the individual work did not contribute to the group effort. We believe that had all the team members used MIDST more proactively during the project, the structure of the nodes (including inputs and outputs) would have helped to structure the dialog about what was being done, perhaps avoiding these problems.

5 CONCLUSION

Our goal in the project described in this paper is to better support coordination in data-science teams by transferring findings about coordination in FLOSS development to the setting of data-science teams. The main contribution of this paper has been to develop a theory of the socio-technical affordances that support stigmergic coordination and to develop a system that supports these affordances. We hypothesize that a tool that offers all of the affordances will enable stigmergic coordination. The evidence (though preliminary) suggests that teams using the tool did in fact have somewhat better coordination with less effort, supporting this hypothesis.

5.1 Limitations of the Study

The evidence suggests that by implementing a set of affordances for shared work, the MIDST tool has improved team coordination. However, the evaluation presented in this paper is only indicative, not conclusive. We plan to continue using the tool with future classes to gather a larger evidence base. We note that the quantitative study results do at least establish the reliability of the scales.
We would also like to use the tool with teams beyond the classroom. For example, we could use the tool with teams that are competing in data-science challenges, such as Kaggle.

5.2 Future System Features

While the system is usable and seemingly useful in its current state, we have plans for additional features. There are a number of small usability improvements, but also four major changes that touch on the affordances developed above.

5.2.1 Encapsulation. At present, MIDST implements modularity incompletely, since it does not enforce encapsulation. All of the code is run in a common R session, which means that a node can use variables that are created in other nodes. At present, use of global variables is not uncommon, e.g., when a user develops a script in RStudio and then copies the code into MIDST nodes. But relying on global variables is a bad idea, as it creates dependencies between nodes that are not visible in the overview. In the worst case, results might be dependent on the order in which nodes are executed. A second problem caused by the lack of encapsulation is that in the current implementation, port names must be unique in the network, which adds a cognitive overload to creating code nodes. Adding better encapsulation should be straightforward.

5.2.2 Genres. Second, MIDST currently does not implement genres in a deep way, in part because the genre repertoire for data science is still emergent. Future work might further develop different genres of analysis tasks to guide users in structuring their analyses. For example, the system could have a template for a typical data-cleaning node or regression node. Such a template could implement a simple default such that the network would be executable immediately (e.g., for a cleaning node simply copying the input data to the output). It could further suggest what actions are typically part of such a node, e.g., including necessary regression diagnostics as part of a regression node.

5.2.3 Tasks vs. Nodes. Third, a major limitation of the current system is the one-to-one mapping of tasks to code nodes. This approach is suitable for the current application where students are developing a new analysis, meaning that the work needed is to develop a node. However, it would likely not be appropriate for a team maintaining an analysis, where a task to be done might affect multiple nodes. We need to rethink the interface and system functionality to represent tasks that are not connected to a single node.

5.2.4 Code Merging. Finally, as noted above, to avoid conflicting changes that might need to be manually merged the system currently allows only the owner of a node to push code changes. We would like to relax this restriction, especially to support tasks that touch multiple nodes, but also to let users more easily make small changes to others' nodes. To do so will require implementing a mechanism to handle multiple change to the same node, including the ability for a user to manually merge conflicting changes.

ACKNOWLEDGMENTS

Omitted for review.

REFERENCES


Wanda J. Orlikowski and JoAnne Yates. 1994. Genre repertoire: The structuring of communicative practices in

Juergen Musil, Angelika Musil, and Stefan Biffl. 2014. Towards a coordination-centric architecture metamodel for


Robert Heckman, Kevin Crowston, Qing Li, Eileen E. Allen, and Yeliz Eseryel. James Howison, Eileen Allen, and Qing Li. [n. d.]. Emergent decision-making practices In Free/Libre Open Source Software (FLOSS) development teams. In IFIP WG 2.13 Working Conference on Open Source Systems.


Appendices

A  SCALES
Via a 5-level Likert scale, where the respondents reported how much they agreed with each of the items in the following scales.

Shared Understanding (Cronbach Alpha: 0.92)
• My team members had a shared understanding of the project and its goals
• My team members had a detailed understanding of each component/module
• My team members had a clear understanding of who was doing which task
• My team members had a common understanding of what inputs were important
• My team members had a common understanding of which analyses were important

Tool Visibility Facilitation (Cronbach Alpha: 0.84)
• The tools I used for this project helped our team share the code across our team
• The tools I used for this project helped our team coordinate our work
• The tools I used for this project helped me understand what others were doing
• The tools I used for this project helped our team divide the project work

Coordination Effort (Cronbach Alpha: 0.86)
• My team spent time planning how it would do the project
• My team discussed who would do what for the project
• My team discussed how long each task would take

Workload Fairness (Cronbach Alpha: 0.85)
• I think the work was divided fairly on this project
• Everyone on the team did their fair share of the work

Perceived Coordination (Cronbach Alpha: 0.92)
• My team members coordinated our actions for this project
• My team members made decisions to maximize our overall team performance
• My team members had a clear understanding of how our tasks should be coordinated
• Everyone on my team knew the status of all the tasks being done by the team
• Everyone on my team understood the details of the tasks being done by the team.

Team Output Issues (Cronbach Alpha: 0.79)
• My team members experienced problems completing work due to delays from other team members
• There was duplication of work (across team members)
• There were situations of unusable task output (from members of my team)