Infrastructor: Flexible, No-Infrastructure Tools for Scaling CS

Daniel W. Barowy and William K. Jannen
[dbarowy,jannen]@cs.williams.edu
Williams College

ABSTRACT

Demand for computer science education has skyrocketed in the last decade. Although challenging everywhere, scaling up CS course capacities is especially painful at small, liberal arts colleges (SLACs). SLACs tend to have few instructors, few large-capacity classrooms, and little or no dedicated IT support staff. As CS enrollment growth continues to outpace the ability to hire instructional staff, maintaining the quality of the close, nurturing learning environment that SLACs advertise—and students expect—is a major challenge.

We present Infrastructor, a workflow and collection of course scaling tools that address the needs of resource-strapped CS departments. Infrastructor removes unnecessary administrative burdens so that instructors can focus on teaching and mentoring students. Unlike a traditional learning management system (LMS), which is complex, monolithic, and usually administered by a campus-wide IT staff, instructors deploy Infrastructor themselves and can trivially tailor the software to suit their own needs. Notably, Infrastructor does not require local hardware resources or platform-specific tools. Instead, Infrastructor is built on top of version control systems. This design choice lets instructors host courses on commodity, cloud-based repositories like GitHub. Since developing Infrastructor two years ago, we have successfully deployed it in ten sections of CS courses (323 students), and over the next year, we plan to more than double its use in our CS program.

KEYWORDS

capacity scaling, github, course workflow, enrollment crunch

ACM Reference Format:
Daniel W. Barowy and William K. Jannen. 2020. Infrastructor: Flexible, No-Infrastructure Tools for Scaling CS. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366905

1 INTRODUCTION

Over the last ten years, the number of students taking computer science courses has dramatically increased. A recent CRA report notes that in large CS departments, enrollments have grown by an average of 185%. Growth is even higher in small[1] departments, averaging 216% [6]. Our experience at a small, liberal arts college (SLAC) mirrors this trend. Despite active measures to shape enrollments, the number of graduating CS majors has grown by 348% over the last ten years. In contrast, the number of faculty in our department, including visitors, has grown by 40% in that same period.

Maintaining high teaching standards in such a climate is challenging, and our department has considered a large number of interventions, ranging from the hiring additional teaching staff to curricular changes, to accommodate the larger number of students. This paper addresses one critical workload reduction opportunity that complements other efforts: reducing the administrative overhead associated with managing assignments.

In this paper, we identify and address an important set of administrative tasks whose time and effort costs should be *constant time per assignment* regardless of the size of the class.

Assignment distribution: providing students with assignment handouts, starter code, and data sets.

Assignment collection: obtaining the most recent (or the most recent before the deadline) assignment version.

Distribution to grading staff: delivering collected assignments to teaching assistants or other grading staff.

Collating feedback: collecting feedback from course staff and recording grades into a gradebook.

Returning graded assignments: returning grades and other feedback to students.

Many of the tasks enumerated above do not contribute directly to the educational value of a course, but in our experience, they take a substantial amount of time. While learning management software (LMS) can address some of this cost, the fact that most LMS systems are monolithic, closed software means that no one LMS can likely address all problems "off the shelf." Importantly, the fact that LMS systems are typically administered by a central IT department (as is the case at our institution) means that LMS systems are not extensible or customizable by end users, even though as computer scientists we possess the requisite skills to do so. Many customizations, like anonymous grading and automated feedback, will likely be added to an IT department's "long tail" of feature requests that are never implemented.

Finally, any solution should consider the needs and limitations of small academic units. Faculty at small institutions often administer custom software without the support of a central IT department.

This paper makes the following contributions:

- we identify a set of generic administrative tasks whose overheads can largely be eliminated;
- we introduce Infrastructor, a set of freely-available, open-source course management tools designed to eliminate those

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366905

[1] The CRA labels units with fewer than 25 full-time faculty as "small."

overheads and to run on inexpensive commodity version control systems hosted in the cloud;
- we discuss a design philosophy that anticipates the "long tail" of features we may never anticipate ourselves; and finally,
- we reflect on two years of successful deployment of the Infrastructor tools, as well as future opportunities for further time and effort savings.

2 INFRASTRUCTOR OVERVIEW

In this section, we describe the Infrastructor design philosophy before giving an overview of the Infrastructor workflow and tools. We use our Data Structures course (colloquially, *CS2*) as a running example. We conclude by discussing adaptations of Infrastructor to other courses.

Infrastructor philosophy. LMS systems are monolithic, all-or-nothing pieces of software. Although instructors may opt into or out of specific features, users cannot modify pre-defined behaviors or design new features for specific course offerings. Instead, we believe that tools should be simple and special-purpose. Complex workflows should be achieved by composing these simple tools. Instructors should be able to insert new functionality *without needing to modify existing code*. This philosophy, borrowed from UNIX, allows Infrastructor to adapt to any workflow [14].

Infrastructor tools are broken into a set of small utilities that each perform a single step of an Infrastructor workflow. If a target workflow cannot be satisfied by an existing combination of tools, tools may be replaced or modified, and new tools may be added. It is commonly the case that either the output of one Infrastructor tool is used as the input to another tool or that a side effect of executing one tool advances the workflow state in preparation for the next Infrastructor tool.

Infrastructor design goals. A successful workflow should handle failures and disconnections gracefully. Whenever it makes sense, Infrastructor tools are *idempotent*. That is, executing a command multiple times produces the same effect as a single successful execution of that tool. Since all commands are also fail-fast [9], instructors know when and how to fix a problem before proceeding. Finally, Infrastructor gives instructors the ability to atomically control the visibility of student repositories even though there may be multiple steps involved in preparing them, e.g., adding starter code.

Infrastructor generic tasks. When developing the Infrastructor tools, we identified a set of tasks that are common among the courses that we teach. By generalizing course workflows to a set of generic administrative tasks, we aim that Infrastructor be helpful in both "laboratory" and "problem set" courses. We also include best practices as sensible Infrastructor defaults. For instance, submitted work is anonymized for blind grading by default, but the degree of anonymization is configurable.

2.1 CS2 Context

Data Structures (colloquially, *CS2*) is a semester-long course comprised of weekly "lecture" meetings and a mandatory "laboratory" meeting. During lab, students begin work on their (typically) week-long programming assignments in a structured environment. Laboratory programming assignments are designed to require significantly more time than the duration of the laboratory meeting itself.

Students submit work at assignment deadlines, but late submissions are often accepted. Completed assignments are distributed to teaching assistants and other course staff in order to perform an initial correctness check and to assign provisional grades. Course instructors then complete a final pass through the assignments to standardize grades, to confirm adherence to assignment rubrics, to correct grading mistakes, and to supplement comments.

Finally, graded assignments are returned to students. Students may ask questions about assignments and request grading clarifications. Students may also correct and resubmit certain assignments. When plagiarism is suspected at our institution, students are not alerted directly; instead, all evidence is assembled by the instructor and submitted to college administration. For these reasons, it is essential that instructors retain original copies of submitted work.

Although we strive to make the weekly student experience uniform, there are factors that make this difficult. Many students elect to use their own personal computers, instead of the shared computing resources provided by our department, so that they may complete assignments at the location of their choice. Collaboration policies differ by assignment; for both pedagogical and practical reasons, some assignments are completed individually and others in pairs. Further, some assignments span multiple weeks, though those assignments typically contain weekly milestones and "progress checkpoints" to ensure that steady progress is made. Checkpoints vary in specificity and in the rigidity with which they are enforced.

2.2 Generic Task Workflow

At a high level, all coursework is (1) assigned by the instructor, (2) completed by students, (3) submitted by students (or collected by course staff), (4) graded by course staff, and finally (5) returned to students. Although the specifics of this process vary by course and assignment, all courses share these tasks. Therefore they form the basic framework of an Infrastructor workflow.

In the rest of this subsection, we explain each Infrastructor tool, and we explain how our CS2 course uses those tools. Figures 1 and 2 illustrate the workflow graphically.

Step 1: `gen-config`. All Infrastructor tools are governed by an assignment-specific configuration file that controls, among other things, the way that assignments are named, where collected assignments are stored and archived, how assignments are distributed to course grading staff, and how, if at all, collected assignments are anonymized for blind grading. Thus, the first step in the Infrastructor workflow is to create a configuration file.

The `gen-config` tool generates a class-specific, per-assignment configuration file from an assignment template and a text file that lists student teams. Teams are described by a comma-separated list of students, one team per line; individual assignments contain groups of one student. To facilitate randomized teams, the `random-groups` utility (not shown) creates and outputs random student pairs subject to constraints (e.g., `studentA` and `studentB` should not be partners). An example template, list of student teams, and configuration file are shown in Figure 3. We created all CS2

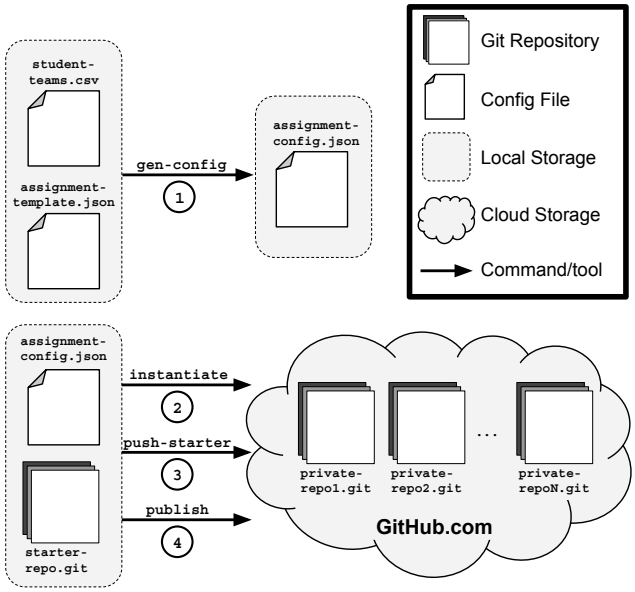

Figure 1: Example CS2 assignment distribution workflow using Infrastructor tools. After generating a per-assignment configuration file (`gen-config`), Infrastructor tools can create private assignment directories (`instantiate-repos`), possibly populate those directories with starter files (`push-starter`), and make the assignment directories accessible by student teams (`publish-repos`).

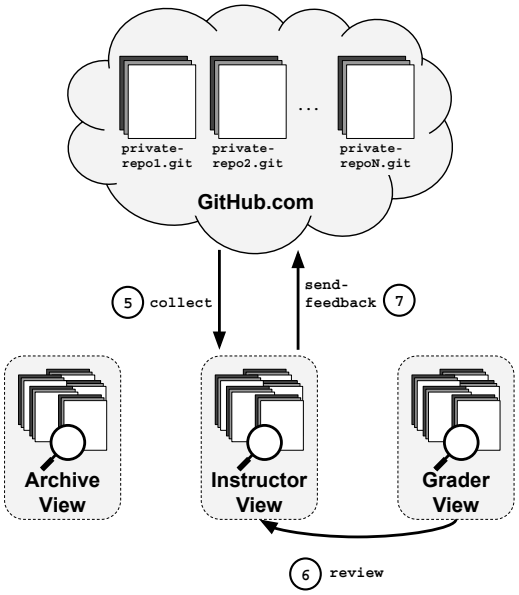

Figure 2: Example CS2 assignment collection/grading workflow using Infrastructor tools. The `collect` tool creates three distinct assignment views, optimized for different tasks. When course staff finishes grading anonymized assignments, the `review` tool integrates feedback from the Grader View into the Instructor View. Finally, the `send-feedback` tool distributes assignment feedback to private student repositories in the form of a "Pull Request".

assignment templates at the start of the semester, and every such assignment we reuse requires little effort.

Step 2: `instantiate-repos`. Private assignment folders are instantiated and named based on their (eventual) student owners. When using the `GitHub` backend, the `instantiate-repos` utility does not consume local storage. Instead, private repositories are created on the `GitHub` cloud platform within the owner organization. After running `instantiate-repos`, repositories are empty and visible only to organization owners.

Step 3 (optional): `push-starter`. Many assignments are prepackaged with starter files. The `push-starter` utility initializes each private repository as a carbon copy of an example repository. In CS2, we also use this tool to provide assignment handouts, documentation, hyperlinks to external resources, and reflection questions associated with laboratory assignments. After running `push-starter`, each private repository specified in the configuration file is populated, but not yet visible to student owners.

Unfortunately, errors in starter files sometimes go undetected by the instructor before release. To correct files later, instructors can push changes on a new branch, as the `push-starter` utility takes an optional `branch` parameter. Students are asked to incorporate fixes using `git`'s built-in `merge` functionality. We consider fluency with merging to be an essential collaboration skill.

Step 4: `publish-repos`. The `publish-repos` tool makes owner-private directories accessible to designated students. We typically call `publish-repos` at the time a CS2 assignment is posted.

Student work. As soon as an assignment directory is published, a student may commit changes to their designated repository. Since the Infrastructor workflow is built on top of the `git` version control system, student progress is preserved in a timeline of self-defined assignment milestones. At any point, students may request that a version of their assignment be reviewed by an instructor, who can access private student files remotely, either by retrieving the latest commit using `git` or by using `GitHub`'s web interface.

Step 5: `collect`. Assignment collection requires no action on the part of a student. The `collect` tool simply retrieves the entire assignment history for each student directory, and displays the most recent snapshot. By default, the `collect` tool indexes submissions under three distinct *views* (directories); each view serves a different purpose and therefore imposes a different organizational scheme:

- *Archive View: An archive of the assignment timeline, including identifying information, used for later reference.* Archived assignments for an entire course are organized first by student name then by assignment. For example, if students Alice and Bob work individually on HW1 and in a partnership on HW2, archive entries will be stored in `archives/alice/hw1-alice/`, `archives/alice/hw2-`

Assignment Template (JSON)

```
{
  "anonymize_sub_path": true,
  "archive_path": "/local/courses/cs2-archive",
  "assignment_name": "hello-world",
  "course": "cs2",
  "feedback_branch": "assignment-feedback",
  "github_org": "williams-cs",
  "hostname": "github",
  "repo_excludes": [ "*.class", "*.o" ],
  "starter_repo": "/local/code/hello-world-starter",
  "submission_path": "/local/courses/cs2-submissions",
  "ta_path": "/shared/courses/cs2-TAs",
  "TAs": [ "alex", "jesse", "riley" ]
}
```

Student Teams (CSV)

```
angus,holstein
jersey,highland
longhorn
shorthorn,wagyu
```

Assignment Config (JSON)

```
{
  "anonymize_sub_path": true,
  "archive_path": "/local/courses/cs2-archive",
  "assignment_name": "hello-world",
  "course": "cs2",
  "feedback_branch": "assignment-feedback",
  "github_org": "williams-cs",
  "hostname": "github",
  "repo_excludes": [ "*.class", "*.o" ],
  "repository_map": {
    "angus": "cs2-hello_world-angus-holstein",
    "highland": "cs2-hello_world-highland-jersey",
    "holstein": "cs2-hello_world-angus-holstein",
    "jersey": "cs2-hello_world-highland-jersey",
    "longhorn": "cs2-hello_world-longhorn",
    "shorthorn": "cs2-hello_world-shorthorn-wagyu",
    "wagyu": "cs2-hello_world-shorthorn-wagyu"
  },
  "starter_repo": "/local/code/hello-world-starter",
  "submission_path": "/local/courses/cs2-submissions",
  "ta_path": "/shared/courses/cs2-TAs",
  "TAs": [ "alex", "jesse", "riley" ]
}
```

Figure 3: Example Infrastructor configuration files. Given an assignment template and a list of student teams, `gen-config` produces a per-assignment configuration file to be used by subsequent Infrastructor commands. Notably, the configuration file contains a student-to-repository map, highlighted in red.

`alice-bob/`, `archives/bob/hw1-bob/`, and `archives/bob/hw2-alice-bob/`.

- *Instructor View: An (optionally) anonymized version of each assignment timeline for instructor review.* Instructor copies are sorted first by assignment then by student team. For example, a single `submissions/hw1/` directory stores assignment directories for each team. By default `anonymize_sub_path=true`, so individual directory names are anonymized, but all version control information is maintained in a hidden directory.
- *Grader View: A completely anonymized directory containing the most recent version of an assignment, used for grading.* Assignments are distributed, round robin, to designated course grading staff. All `git` information is stripped. These files can be shared with TAs in any manner instructors find convenient (e.g., email, shared filesystem, etc.), as long as marked-up files are placed back in their original locations.

Locations for these views are specified in the configuration file using the `archive_path`, `submission_path`, and `ta_path` keys.

Step 6: `review`. After comments are made to anonymized assignments in the *Grading View*, the `review` tool applies those edits to the version-controlled *Instructor View* so that feedback can be audited. Instructors may further comment assignment files or make corrections. When satisfied with the state of an assignment's feedback, the instructor simply continues to the next assignment. Feedback review is disentangled from feedback distribution so that instructors may revisit assignments before releasing feedback to students.

Step 7: `send-feedback`. The `send-feedback` tool takes all assignment files, with feedback, and preserves them in a snapshot for students. Snapshots are released in a format optimized for comparison against submitted student work. `git` includes tools for showing file changes for review (`git-diff`), and `GitHub` provides a convenient web interface called a *pull request* [3] that shows changes in a variety of formats [2]. `GitHub` pull requests also provide tools for dialogue so that feedback can be clarified on request.

2.3 Adapting Infrastructor to Other Courses

Although we began the Infrastructor project with laboratory courses in mind, our goal was to develop a single set of tools for use across our curriculum. In addition to laboratory courses, we teach theory courses where students primarily complete work in the form of regular LaTeX-formatted problem sets. Surprisingly, using Infrastructor eliminated several administrative challenges.

Since the `push-starter` tool lets us seed a repository with files, we can eliminate inconsistent file naming conventions among students. However, we can go further. By creating problem set templates with one page/file per problem solution, we can more easily script the distribution of work to our TAs. For example, all instances of problem one can be graded by the same TA for consistency.

3 FOUNDATION

Infrastructor is built on top of version control. Version control offers a small, but important, set of primitive functions that make it easy to construct reliable workflows.

In general, we are ambivalent about the choice of version control implementation. We built Infrastructor on top of `git` for pragmatic reasons: `git` is widely available, free and open source software, and it is the *de facto* version control system used by industry today. Likewise, we use `GitHub` because it is free of charge for educators and it offers a convenient web interface for students. However, any version control system that provides the the ability to *atomically* create repositories, and to *atomically* add, update,

replace, and check the status of a file provides the necessary foundation for our tools [17].

Infrastructor provides two guarantees to instructors. First, Infrastructor ensures that updates to a course workflow are *consistent*. Secondly, Infrastructor maintains the *confidentiality* of student work and the *anonymity* of student identities.

3.1 Consistency

Infrastructor provides an important safety guarantee to all commands: that running the same command with the same inputs yields the same outputs. This guarantee, called *idempotence*, is critical to successfully deploying a course workflow.

Idempotence. Let r be a version control repository, which is a map from filenames to file contents (a byte array), and let o be a version control operation. Let command c be a sequence of n operations $o_1, \ldots, o_n$. Atomically applying all n operations in sequence to repository r_0 yields repository r_n. More formally, $r_n = o_n(\ldots o_2(o_1(r_0)))$. We say that repository r_n is *consistent with* c. Since any atomic operation in a distributed application can fail, such failure may leave the repository *inconsistent with* c, i.e., not in state r_n. For example, when operation o_i of c fails, it leaves some operations only partially applied, such that $r_{i-1} = o_{i-1}(\ldots o_1(r_0))$.

A command is *idempotent* if and only if application of command c is consistent, given any r_i, where $0 \leq i \leq n$. Idempotence ensures that, when a failure occurs, the user can rerun c, and when c successfully terminates, the repository is guaranteed to be consistent. For example, if the network fails partway through an `instantiate-repos` call, running `instantiate-repos` again when network access is restored will correctly create all the specified repositories. Importantly, calling `instantiate-repos` more than once will not create duplicate repositories or change the state of any repositories that were already successfully created.

3.2 Privacy

Infrastructor also provides two privacy guarantees: *confidentiality* and *anonymous grading*.

Confidentiality. Although we cannot completely rule out the unauthorized sharing of homework solutions between students, we think that a learning management system should make such sharing difficult or at least inconvenient. Access controls prevent obvious attempts to gain unauthorized access. For example, it should not be possible to for an unauthorized student to `git-clone` another student's repository. Mitigating unauthorized access is important both to maintain academic honesty and to maintain the confidentiality of student work, which is required by law in the United States [7].

Other, less obvious violations can occur, and Infrastructor seeks to prevent these as well. For example, a student may use the `GitHub fork` command to create a copy of their repository. This copy is under their own control. Such a student could then configure their forked copy to bypass the mandatory access controls put in place by the instructor on the original repository. Fortunately, `GitHub` provides a per-repository "allow forking" flag. Infrastructor sets this flag to `false` by default.

Anonymous Grading. Bias in grading has been a cause for concern for educators for decades [10–13]. Our department has consciously moved towards anonymous grading, partly due to our own concerns, and partly because our students have advocated for it. Infrastructor was instrumental in making anonymous grading feasible in our courses, as the practice otherwise incurs substantial administrative overheads. We reserve judgement on the appropriateness of anonymous grading for each circumstance. Instead, Infrastructor gives instructors the discretion to choose the policy they deem most appropriate.

Infrastructor exposes three anonymity levels, a_0: *no anonymity*, a_1:*anonymous to graders*, and a_2: *anonymous to all staff*. If requested, anonymization is applied during the `collect` phase. Anonymization consists of scrubbing repositories of identifying information, like `git` metadata, and renaming local copies of student repositories to pseudorandom identifiers.

Anonymity levels affect the visiblity of files in Infrastructor views (see *Step 5* in Subsection 2.2). a_0 performs no anonymization to any view. a_1 anonymizes work in the *grader view*. a_2 goes an extra step, anonymizing both the *grader view* and the *instructor view*. The *archive view* is never anonymized, since its purpose is to function as a read-only archive of committed work.

We make use of undergraduate teaching assistants in our largest courses. Since students grade their peers, we think it prudent to obscure student identities. We also anonymize student submissions for courses where coursework is graded solely by the instructor. In both cases, we hope that Infrastructor helps us reduce the potential for grading bias.

4 SYSTEM REQUIREMENTS

Infrastructor is written in Python 3 and has two dependencies: the `rsync` file copy tool and the PyGithub library [4, 5]. Although we developed and tested Infrastructor on macOS and Linux, we do not anticipate major obstacles to porting the system to Windows since both dependencies have Windows equivalents. Currently, students can use any computer that has `git` and a web browser, since students do not interact with Infrastructor directly.

We plan to investigate porting Infrastructor to additional version control systems in the future. See Section 7.

5 EXPERIENCES

In this section we discuss our experience with Infrastructor features, from both the instructor and student perspectives. We also discuss how we chose some of the default system behaviors.

5.1 Instructor Advantages

Our primary goal when developing Infrastructor was to reduce the administrative burden of teaching a computer science class. We have used the software with 323 students in ten sections over two years. Six people, including faculty and lab instructors, have utilized Infrastructor for their courses during this period. We have used the tool for core computer science courses, including introductory CS classes, and for systems electives. Although there remain other sources of overhead not addressed by the tool, adoption within our department continues to grow, which we consider a success.

Although a little work needs to be done to create assignment templates the first time Infrastructor is used, this work is amortized when templates are reused. An instructor simply modifies the assignment template for new grading staff (see Figure 3) and reruns the tools. Typically, this takes only a few minutes.

Another advantage are `git` histories. Having an entire timeline allows us to monitor the progress of students. For example, we recently experimented with regular one-on-one code review sessions. Version histories and diffs were an essential part of these interventions. In fact, code reviews would not have been possible without Infrastructor, as the tool eliminated a substantial amount of coordination work previously assigned to our lab instructor.

We also find Infrastructor's archival feature to be useful when handling plagiarism cases. Although this is not a pleasant aspect of teaching, it does appear inevitable. Unusually short version histories are often strongly indicate plagiarism.

5.2 Student Advantages

Infrastructor tools are not visible to students. Instead, students interact with Infrastructor using only `git` and `GitHub`. Students only need a machine with the `git` tool installed and a web browser. Since students do not need a custom environment for submitting their assignments—as used to be common at our institution before we developed Infrastructor—students are free to work in our computer lab or on their own computers.

One aspect of Infrastructor appreciated both by students and faculty is that homework assignments maintain the illusion of a typical software engineering workflow using `git`. Because of this, there is no learning curve for students beyond instruction in `git`. In fact, we now utilize `git` in all classes that have coding components. Using `git` at the introductory level is made more friendly by the BlueJ development environment's built-in `git` support (sometimes used in CS1). Students do not need to actively submit an assignment, since submission is a byproduct of running `git-push`.

Many students reported that `git` helped accommodate conflicting student schedules for group work since they could either pair program in person or collaborate remotely.

`git`'s branching features are also useful. Since instructor feedback is pushed to a separate branch, students have total control over the mainline development in their repositories. Such separation makes it easy to give rapid feedback, including inline comments and code suggestions, without worrying about interrupting student work. We also observe that a small number of students enthusiastically adopt branching, enabling them to experiment or debug without the worry of introducing hard-to-revert code changes.

Finally, `GitHub`'s pull request feature allows students and instructors to carry out conversations about feedback. Students often request clarification on the feedback in a pull request, referring to the diff when discussing misunderstandings.

5.3 Lessons Learned

Here, we discuss some of the hiccups we encountered while developing Infrastructor. These issues informed key design decisions.

Our most important mistake was not thinking carefully about atomicity and failure modes for commands. `GitHub` unavailablity, network problems, and power outages contributed to failures partway through long-running commands more commonly than we anticipated. By requiring that operations be atomic and that commands be idempotent, we have since removed this hassle.

Another surprise arose because visibility of repositories was tightly coupled to their creation. Every now and then, an eager student committed code to their repository *before* we had the opportunity to run the `push-starter` command, causing `git` conflicts that required manual intervention. This prompted us to decouple repository creation, population, and publishing.

Finally, early versions of Infrastructor enforced deadlines for student work by default, ignoring work committed after the deadline. We found that exceptions to deadline policies were sufficiently numerous (e.g., sickness, late homework allowances, etc.) that this introduced more headaches than it solved. Infrastructor now `collects` all homework committed at the time the command is run. We plan to write a separate tool to check for deadline violations.

6 RELATED WORK

Rößling et al. studied the Moodle LMS from the perspective of CS educators and concluded that mainstream LMSs, including Moodle, do not support many activities common in CS education [15]. As a result, Moodle users either string together disconnected tools to perform these activities or do not perform these activities at all. Unlike LMSes, Infrastructor is a set of independent tools that can be composed to support many usage patterns. Features can be added as standalone utilities and incorporated as needed.

Several projects have explored academic workflows on the `GitHub` platform. Teacher's Pet [1] provides a set of Ruby-based tools for creating assignments, populating those assignments with starter files, opening issues, and collecting work. Teacher's Pet is only compatible with the `GitHub` platform. `GitHub` recently deprecated support for Teacher's Pet, introducing `GitHub` Classroom, which natively supports Teacher's Pet functions within the `GitHub` web interface [8]. `GitHub` Classroom suffers from the problems with monolithic systems described above. By contrast, Infrastructor works on `GitHub`, but is adaptable to other platforms and usage patterns because its design philosophy is fundamentally extensible.

7 FUTURE WORK AND CONCLUSION

We plan to continue Infrastructor development. Support for automatic plagiarism detection, with tools like Moss [16], and automated feedback, such as code linting and autograding, are high priorities. We also plan to support alternative platforms, such as `GitLab`, so that instructors who do prefer local hosting can benefit from our tool. Finally, we plan to investigate better support for scaling the number of instructors who use Infrastructor to simultaneously manage the same course. The primary issue is that instructors need to share up-to-date Infrastructor configuration files. Such divisions of labor are common at institutions with traditionally large enrollments.

In this paper, we introduced Infrastructor, a set of flexible tools that largely eliminate administrative overheads for CS instructors. Our experience using Infrastructor across ten CS sections demonstrates that time can be saved in a wide variety of courses and instructional styles. Infrastructor requires no dedicated on-site resources and can be deployed using commodity version control services hosted in the cloud.

REFERENCES

[1] 2016. Teacher's Pet. https://github.com/education/teachers$_{p}$et.

[2] 2019. About comparing branches in pull requests. https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-comparing-branches-in-pull-requests.

[3] 2019. About pull requests. https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests.

[4] 2019. PyGithub: Typed interactions with the GitHub API v3. https://github.com/PyGithub/PyGithub.

[5] 2019. rsync. https://rsync.samba.org/.

[6] CRA. 2017. *Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006.* Technical Report. Computing Research Association, Washington, DC. https://cra.org/data/Generation-CS/

[7] FERPA. 1974. Family Educational Rights and Privacy Act of 1974. 20 U.S.C. §1232g.

[8] GitHub. 2019. GitHub Classroom. https://classroom.github.com. [Online; accessed 31-August-2019].

[9] Jim Gray. 1986. Why Do Computers Stop and What Can Be Done About It?. In *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*. 3–12.

[10] Rema N Hanna and Leigh L Linden. 2012. Discrimination in Grading. *American Economic Journal: Economic Policy* 4, 4 (2012), 146–168.

[11] Edward Lotto and Bruce Smith. 1979. Making Grading Work. *College English* 41, 4 (1979), 423–431. http://www.jstor.org/stable/376285

[12] John M Malouff, Ashley J Emmerton, and Nicola S Schutte. 2013. The Risk of a Halo Bias as a Reason to Keep Students Anonymous During Grading. *Teaching of Psychology* 40, 3 (2013), 233–237.

[13] John M Malouff and Einar B Thorsteinsson. 2016. Bias in grading: A meta-analysis of experimental research findings. *Australian Journal of Education* 60, 3 (2016), 245–256. http://search.proquest.com/docview/1837826758/

[14] Eric S. Raymond. 2003. *The Art of UNIX Programming.* Pearson Education.

[15] Guido Rößling, Myles McNally, Pierluigi Crescenzi, Atanas Radenski, Petri Ihantola, and M. Gloria Sánchez-Torrubia. 2010. Adapting Moodle to Better Support CS Education. In *Proceedings of the 2010 ITiCSE Working Group Reports (ITiCSE-WGR '10)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/1971681.1971684

[16] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[17] Wouter Swierstra and Andres Löh. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/2661136.2661137