λ – An Autograder for Snap*!*

by Michael Ball

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Dan Garcia
Research Advisor

May 13, 2016

(Date)

* * * * * * *

Professor Armando Fox
Second Reader

May 13, 2016

(Date)

Table of Contents

1 Abstract

While visual programming languages are hardly a new development, recent pushes for increased equity and access to computer science have led to a renewed interest and use of visual programming languages. Autograders are a critical component of both in-person and online computer science courses. However, until this point, there haven't been any autograders documented for general-purpose, visual programming development environments. This is a particular shortcoming as introductory courses have scaled to larger numbers of students and online environments. While there are challenges to using autograders, we believe that the instant feedback capabilities, as well as potential time savings for course staff will help us teach a greater number of students.

Over the past year, we have built an autograder, named λ (Lambda), for Snap*!*, a visual blocks-based programming language inspired by MIT's Scratch. The primary motivation for developing the autograder was to run a series of Massive Open Online Courses (MOOCs) on edx.org throughout the 2015-2016 academic year. However, we also wish to use the autograder to better support in-person computer science courses. In Spring 2016, the Snap*!* autograder was used as a part of UC Berkeley's CS10, *The Beauty and Joy of Computing*, a "CS0" non-majors course.

This report describes λ, which consists of a "backend" Ruby on Rails webserver that allows us to use the autograder in a classroom setting, through a protocol called LTI. The backend web application contains a database of questions and test files, while the Snap*!* interface contains new features and a view to present the results of the autograder. Our initial results show the autograder successfully being used in CS10, where the autograder was used to supplement oral lab checkoffs, and on edx.org where the autograder was the primary method for students to receive credit for code, graded both for effort and correctness.

2 Acknowledgements

The work presented here exists due to the contributions of many people and grants from the National Science Foundation (grant 1138596), Google, and edX. Building an autograder would not be possible without BJC, Snap*!* and CS10, which have collectively been supported by hundreds of dedicated students and teachers over the past seven years.

In particular, a huge **Thank You** to the following people who helped make this possible. Thanks especially to Lauren Mock for always being there, and for all the advice and help! Everyone has been incredibly supportive over the past year.

- Dan Garcia for being a mentor for the past five years
- Armando Fox for reviewing this work
- Brian Harvey and Jens Mönig for Snap*!*
- Max Dougherty, Tina Huang, Yifat Amir and Patrick O'Halloran for their incredible work on the autograder
- The CS10 staff for testing unproven technology and for dealing with additional student questions!
    - TAs: Rachel Huang, Adam Kuphaldt, Alex McKinney, Amruta Yelamanchili, Arany Uthayakumar, Erik Dahlquist, Janna Golden, Joseph Cawthorne, Lara McConnaughey, William Tang, Yifat Amir, Andy Schmitt, Steven Traversi, and Victoria Shi
    - Instructors: Justin Hsia and Gerald Friedland

3 List of Figures

4 Introduction

In recent years, the field of computer science (CS), and the broader technology industry, have undergone significant changes, though not simply technical ones. Around the same time that enrollments started booming [1], the National Science Foundation (NSF) posted the original solicitation for *Broadening Participation in Computing* [2]. As a result of the efforts to improve diversity in CS, curricula such as *Exploring Computer Science* [3] (ECS), and *AP Computer Science Principles* [4] (AP CSP) have emerged. Many curricula have started to use visual programming languages (VPLs), also known as blocks-based languages, as their primary tool. As great as these environments can be, they often lack resources and tools found in more traditional environments (such as Python or Java). This lack of infrastructure can make courses more difficult to teach and scale, particularly to entirely online environments.

We aim to help students learning programming by:

- Delivering the best resources to possible to completely remote students, and
- Improving the capability and efficiency of in-person teaching assistants (TAs).

One example of missing resources is the capability to automatically evaluate code, and send the results to another process. Automatic evaluation and distributing the results are central components in an *autograder*. Without an autograder, it can be difficult (if not prohibitively expensive) to teach large online courses. During the 2015-2016 school year, we taught *The Beauty and Joy of Computing* [5] (BJC) as a series of four *massive open online courses* (MOOCs) on edX [6]. BJC is an AP CS Principles course that uses Snap*!* as its primary language. Without an autograder, we did not think that we could fairly give credit to students online. At Berkeley, BJC is offered as CS10, which could use an autograder for Snap*!*.

We developed λ as a system for autograding Snap*!* [7]. λ is composed of two main parts, the Ruby on Rails application server, and a Javascript application that augments Snap*!* with testing, analysis and logging capabilities. While the grading interface was developed for edX, we report on its use in the classroom and some recommendations for future development. We also hope that λ will be useful outside CS10, through the implementation of the Learning Tools Interoperability (LTI) protocol which should allow our system to be compatible with most Learning Management Systems. We have built additional features which should allow flexibility to integrate the autograder into a variety of classroom environments. These features are described in the chapter on implementation.

4.1 Snap*!*

Snap*!* is a blocks-based, drag-and-drop programming language inspired by MIT's Scratch [8], but adapted for university-level courses by including features such as first-class lists and custom functions (blocks). Snap*!* is implemented as a web application in JavaScript, which makes it compatible with mobile devices. Figure 1 shows a basic drawing script being executed.

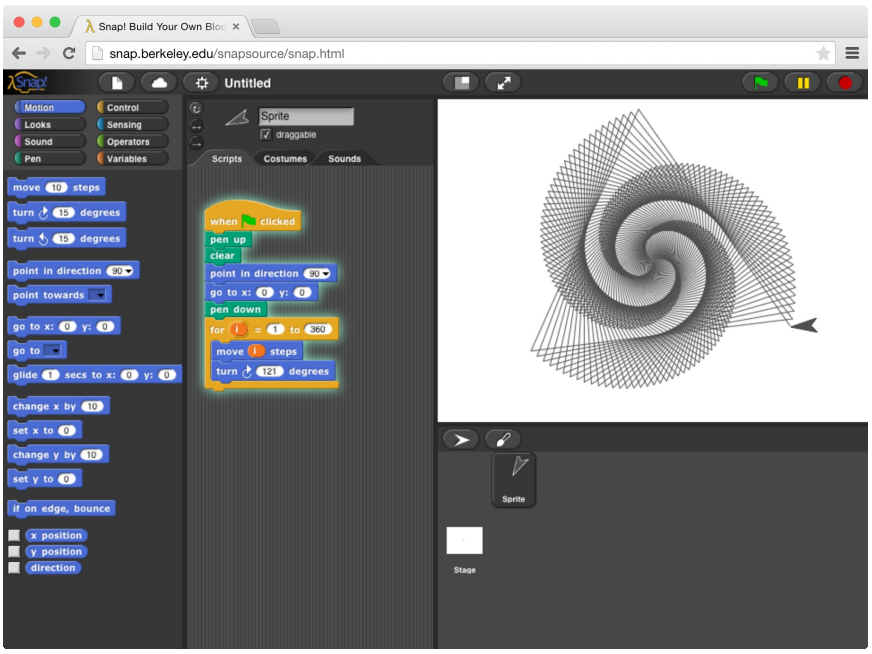

Figure 1: *An example Snap! program.*

4.2 The Beauty and Joy of Computing

The Beauty and Joy of Computing (BJC) is an introduction to computing designed to broaden participation among underrepresented groups. The primary language used is Snap*!*, and many of the exercises in the course have visual outputs. The course covers functions and abstraction, recursion, higher order functions (lambdas) and many other topics. Many of the examples and exercises in BJC have multiple correct paths to implementation, which can be a challenge for autograders to handle.

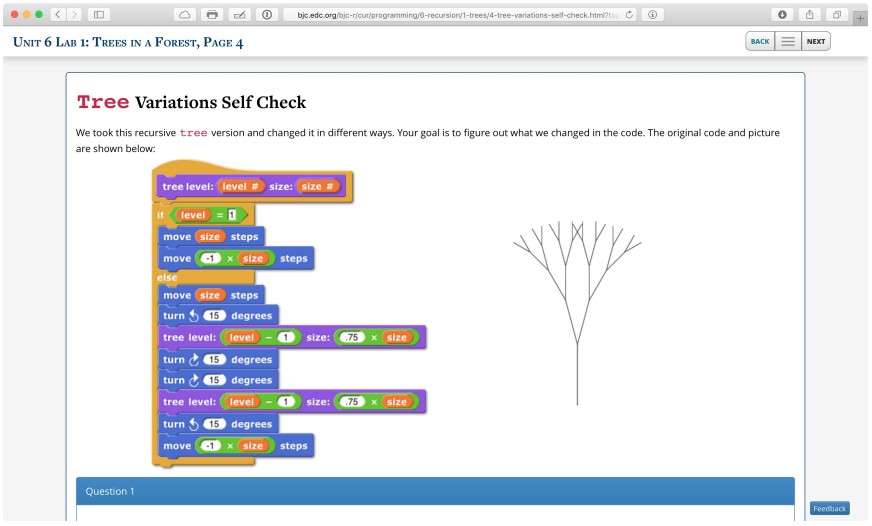

Figure 2: *A typical example of BJC curriculum which includes graphical output.*

4.3 CS10

CS10 [9] is UC Berkeley's offering of BJC. Currently, CS10 is offered every semester to 200-300 students. Like many other introductory courses, CS10 relies heavily on laboratory sections as the primary method for students to learn to program. λ will be used in CS10 during lab sections to give students better feedback as they are working and to give credit for the assigned lab work. This year, we were able to trial the system for three of the 14 labs that students completed in Snap*!*.

4.4 Correctness and Effort

While designing the autograder, we considered two primary types of grading: *correctness* and *effort*. Correctness is simple: Does is given block (or script) do what the instructions say? Effort is more open-ended, but the goal is to be able reward students for trying to complete a problem, as well as to encourage experimentation with Snap*!*'s many features. In both these cases, we would like the autograder to be robust to different correct solutions, or the many different ways students can demonstrate effort.

5 Related Work

In building and designing an autograder, there is surprisingly little work published about autograders, and even less so about those for visual programming languages. While there are many examples of autograders for languages like Python and Java, there are only two that we are aware of for blocks-based languages, which are both very recent developments.

5.1 Code.org CodeStudio

Code.org's CodeStudio [10] is an online interactive environment based around the open-source Blockly environment [11]. CodeStudio includes custom feedback, but the vast majority of the problems are in a highly restrictive space where students are given only a limited set of blocks. While CodeStudio is open source [12], the autograder and feedback methodologies are not documented.

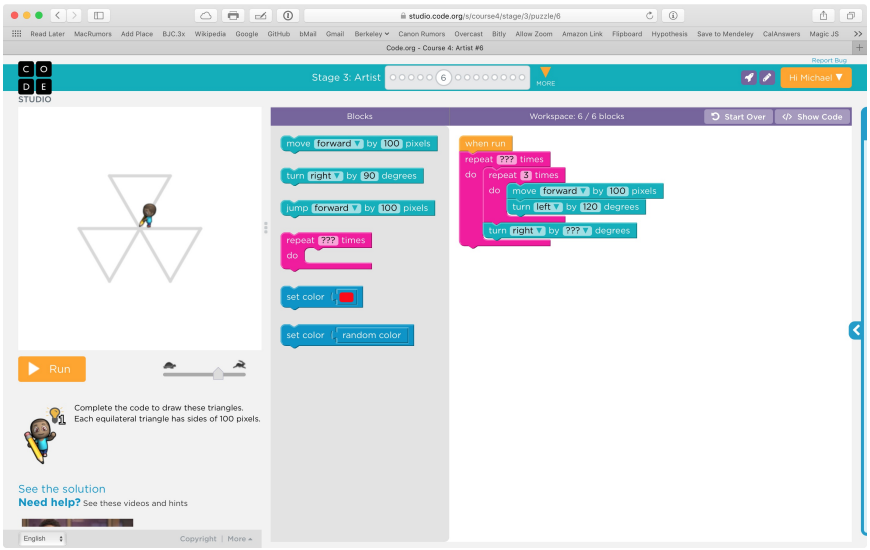

Figure 3: *A typical example of a CodeStudio problem that gives students only a few blocks to work with, and has a fairly constrained solution space.*

5.2 ITCH: Individual Testing of Computer Homework for Scratch Assignments

ITCH is an autograder for Scratch recently presented at SIGCSE 2016 [13]. However, ITCH's method of autograding is through an instructor-initiated script which should be run after all assignment submissions are collected. This is a fairly common scenario for typical code autograders, but not a model we would use because our goal is instant feedback. ITCH takes a similar method to our Snap*!* autograder by 'spoofing' user inputs when requested, but it shares many of the same limitations, like a lack of feedback for graphical output.

6 Design Principles

Educational design principles helped influence the design of the autograder and the direction for creation of exercises. While these design principles helped influence features and the user interface of the autograder, they are also extremely important as a guideline for how instructors and content authors should think about writing exercises and implementing them in the classroom.

6.1 Dual Modalities

A central challenge faced when considering development of the autograder was what to prioritize: Feedback or Grading? While both features are necessary for each other, there is an acute tension between a tool which is primarily motivated by providing open-ended feedback, and one that is designed to provide grades. The goal of this section is to consider the best possible interface we could give to students to help them improve their programming skills and complete lab exercises.

A key aspect of this tension is how to handle the idea of correctness. In an introductory computer science course, we are often lenient with small differences in output from a program. (For example, Snap*!* allows students to use both traditional arrays and linked lists, but their visual output is the same. If test authors are not careful, it is easy to mistake one type of data as incorrect when it should be accepted.) While it is often possible to account for these differences when writing test cases, it can be significantly difficult. We need to make sure that when these tools are used for *grading*, they do not cause students unnecessary stress or frustration.

6.2 Learner-Centered Design

Learner-centered design (LCD) is a design principle adapted from user-centered-design (UCD) [14] [Citation not found] [15]. Both LCD and UCD design principles start by establishing attributes about the user's goals. For LCD there are 4 main attributes:

> • Learners do not possess the same domain expertise as users.
>
> • Learners are heterogeneous.
>
> • Learners may not be intrinsically motivated in the same manner as experts.
>
> • Learners' understanding grows as they engage in a new work domain.

While languages like Snap*!* are designed to be easier to learn, they do not necessarily employ LCD principles because they are still intended as a general-purpose programming environment. We can make sure our autograder takes into each of these principles:

Domain expertise: Programming languages have to show error messages that could make sense in any situation. Unfortunately, this means they usually fall to the lowest common denominator type cases, and do not provide any contextual information to the use. In Snap*!*, it is not uncommon to see a message that is similar to `Type Error: expecting a list but got number`. We can improve upon these messages by showing students hints which are specific to the problem at hand.

Heterogeneity: This is one of the harder aspects for the autograder to handle. Not everyone approaches problems in the same way. Our general approach is to try to be as lenient as possible (while still ensuring correctness) when writing test cases. We have spent lots of time considering how authors should handle different formats of output so that we try to avoid nit-picky errors.

Motivation: We try to motivate users by carefully choosing how we present the tool and the results. In class, and in the text which appears on screen we try to downplay the idea of grades or errors and instead focus on helping students improve.

Changing Understanding: Dynamically capturing a user's understanding is incredibly difficult to do. At this point, we are not able to dynamically adjust exercises or feedback presented, but we have planned out possible methods for doing so. Currently, the best way for us to achieve this is to have Teaching Assistants (TAs) and instructors, who are conscious of students needs, recommend different problems for students to practice with.

6.3 Knowledge Integration

Knowledge integration (KI) is a framework for approaching how students should synthesize information [Citation not found] . The KI framework has four components to orangize ideas:

- **Adding Knowledge** involves bringing in new ideas that students have not seen before.
- **Eliciting Ideas** is the process of critically examining ideas students already know.
- **Distinguishing Knowledge** asks students to take multiple ideas and figure out how they fit together; whether they are compatible or not.
- **Reflecting** is the process of drawing conclusions from what students have learned.

We used KI as a basis for writing the feedback messages that students are shown through the autograder. The goal is to focus primarily on the eliciting knowledge and distinguishing knowledge components. We wanted to focus on these two pieces because there are many common computer science problems which can be viewed through this lens. Systematically debugging code follows a process of eliciting knowledge when trying to figure out why something is broken. Distinguishing knowledge (such as the differences between two kinds of loops, or recursion and iteration) is a natural process for programming.

We chose not to use the *adding knowledge* component because we do not currently have the autograder setup to give good feedback when students are doing exploratory work (where they would be most likely to uncover new concepts). However, these types of messages will likely appear in future versions. Similarly, while *reflecting* is a valuable step, we do not have the capability to collect nor give feedback to open-ended reflection questions.

However, writing proper KI messages proved challenging in the current setup. The initial version of the autograder was designed more around presenting the results of test cases, than it was for longer forms of feedback. (This is one area for improvement.) Furthermore, trying to follow KI occasionally led to messages that did not necessarily fit within the rest of the BJC curriculum as it was not designed around the KI framework.

6.4 "TA-Centered Design"

Though this is certainly lower on the priority list than *learner*-centered design, we make a point to describe *TA*-centered design, and why this matters for the tools we build. Teaching Assistants (TAs) and instructors, are critical users of the infrastructure in courses. They need to be able to easily update and create content, handle grades and so on. The longer or more difficult these tasks are, the less time TAs have to spend helping students learn.

When considering TA-Centered Design, a TA is much more like a typical user in UCD than a learner in LCD, but there are many ideas that should be specifically recognized for TAs:

- TAs are often lacking pedagogical content knowledge (PCK). PCK is making the distinction between knowing how to program, and how to teach programming. TAs could use guidance in applying good pedagogy.
  - The admin dashboards built into λ give TAs more insights than they currently have about how students are performing and how often they are completing the lab work. While the dashboards have a ways to go in functionality, this is an improvement and gives TAs a reason to keep using the system.
  - The test case authoring interface should be adapted to make it effortless to write consistent and detailed feedback.

- While TAs are motivated to *teach* they are not always motivated to complete the extra work required of them, such as grading or writing assignments.
    - Test cases need to be easy to write and upload.
        - The Implementation chapter describes the problems with our initial approach using edX's tools.
        - The Future Work chapter describes how we can improve the experience of writing autograder test files to lower the barrier.
- Often TAs are not experts in the tools they are required to use to accomplish their teaching duties, such as LMSs and grading systems such as λ.
- TAs, like most users, have a limited amount of time to complete their work.
    - Limit (or automate) repetitive tasks, especially ones that involve configuration.
    - The ability to automatically upload grades for students is a huge time saver which allows TAs to focus on more important tasks, and allows students to stop worrying about the status of their grades.

While these three ideas may seem obvious, they are important to recognize if our work is to be used beyond our initial test implementation. Then, we need to consider how TAs will use λ. The success of a new autograder, even if it is beneficial for students requires TAs to be comfortable configuring and writing new autograder tests.

Ultimately, we're are trying to recognize that TA's (or individual instructors) are already limited by time. Making test cases as easy to write as possible is a necessary part of the process.

7 Interface Walkthrough

λ is composed of two parts: a webserver and a Snap! interface with autograding capabilities. In the current implementation, only instructors need to use the webserver, though in the future there may be functionality added for students. (See the future work section.)

7.1 Web Application

The basic web interface (figure 4) presents a list of problems to try. This list is public, so anyone can attempt any problem, but instructors are expected to embed specific questions with their directions. Users can click on a link to work on a particular question.

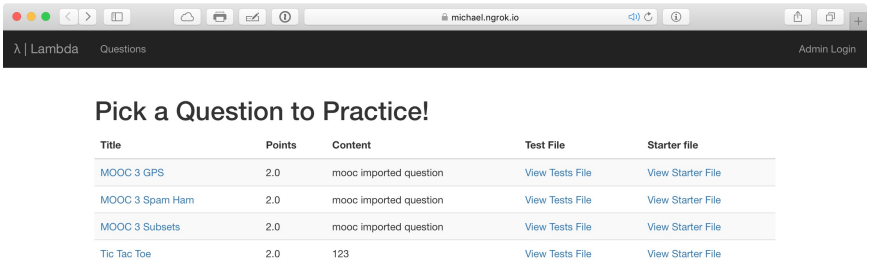

Figure 4: *The initial page is a list of questions to try.*

The rest of the interface options come after the user has logged in with an admin account (figure 5).

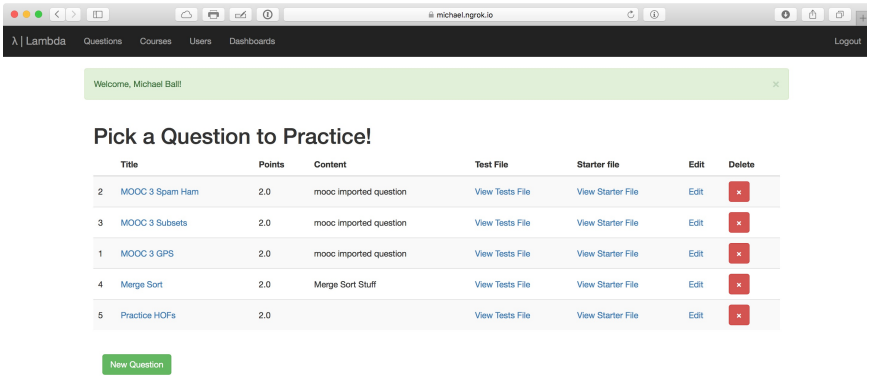

Figure 5: *Administrators have additional functionality.*

These features are:

- **Creating / Editing Questions** - A question contains a test file, a points value, and a starter file.
- **Creating / Editing Courses** - Courses describe connections to the LMS. Each course has a key and some policy settings (figure

6).

- **Creating / Editing Admin Dashboards** - Admin dashboards provide the status of student performance. This initial version is based entirely on custom SQL queries, but they can be powerful. These dashboards were a significant benefit in doing analysis for this report and were shared with TAs during the course (figure 7).

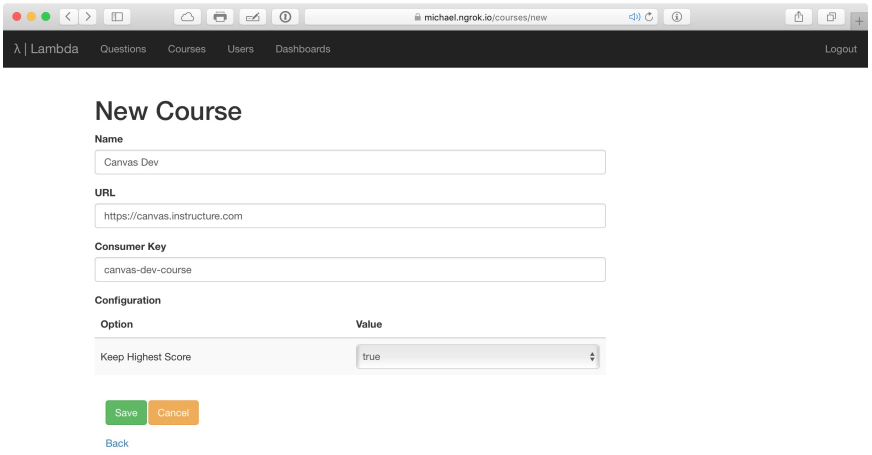

Figure 6: *Creating a new course is a simple action which requires little information.*

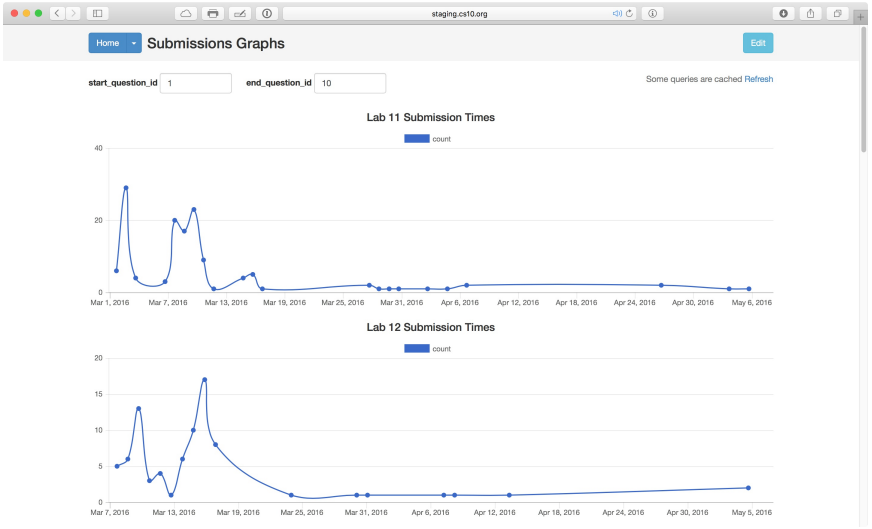

Figure 7: *A dashboard showing the first two labs the submission times for autograder requests for.*

7.2 The Snap*!* Interface

The autograder augments the Snap*!* interface with a few basic tools. The initial version of the autograder included a button and "hamburger" menu, that were designed to appear integrated into Snap*!*. However, the apparent seamlessness was actually more confusing because the controls never fit within the rest of Snap*!*.

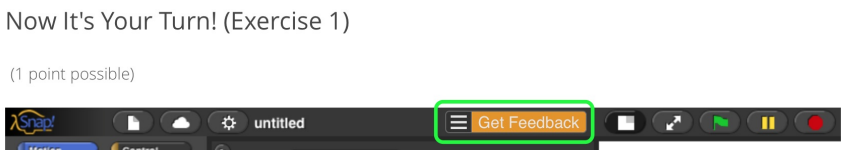

Figure 8: *The initial (edX) version which had a heavily integrated feedback button.*

The updated version more clearly separates the autograder controls from Snap*!*, and gives us more room to expand functionality in the future. A problem title will always be displayed in the Snap*!* interface (compared to outside the window in the edX version), which was a small but important change because the new setup allows for displaying Snap*!* in a separate tab from the question instructions.

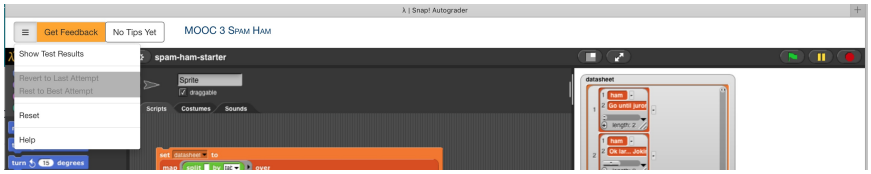

Figure 9: *Updated controls for the autograder showing a dropdown menu. (The controls for reverting submissions are greyed-out.)*

Both interfaces have the following features for students:

- The "*Get Feedback*" button runs a set of autograder tests on the code, which are presented when tests are complete.
  - The color of the status bar (the button in the edX version) changes color based on the question's state:
    - Green: All tests are passing
    - Orange: The student has modified their code and tests should be re-run.
    - Red: At least one test is failing.
- *Restore Best Submission*
- *Restore Last Submission*
  - These two features encourage students to experiment and re-write code without any fear that they might hurt their scores or lose work. Every time the "Get Feedback" button is clicked, a submission is recorded.
- *Reset*
  - Restores the current Snap*!* project to the state of the starter file.
- *Help*
  - Displays a set of tool tips over autograder-specific elements.
- *Show Previous Results*
  - This is a new option which will present the feedback of the previous tests without re-running them. This allows students to review mistakes, and should discourage "autograder-driven development".

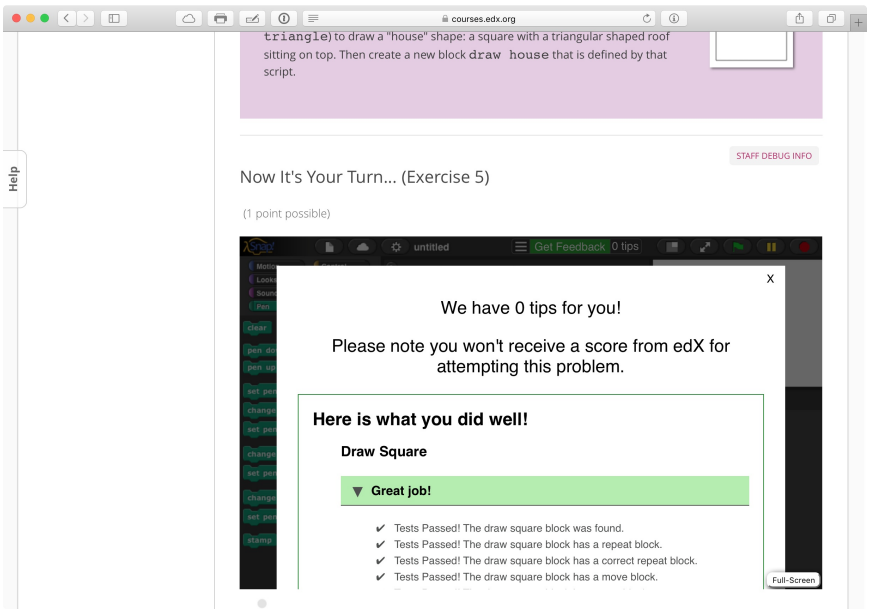

Figure 10: *An example of the feedback presented when everything is correct.*

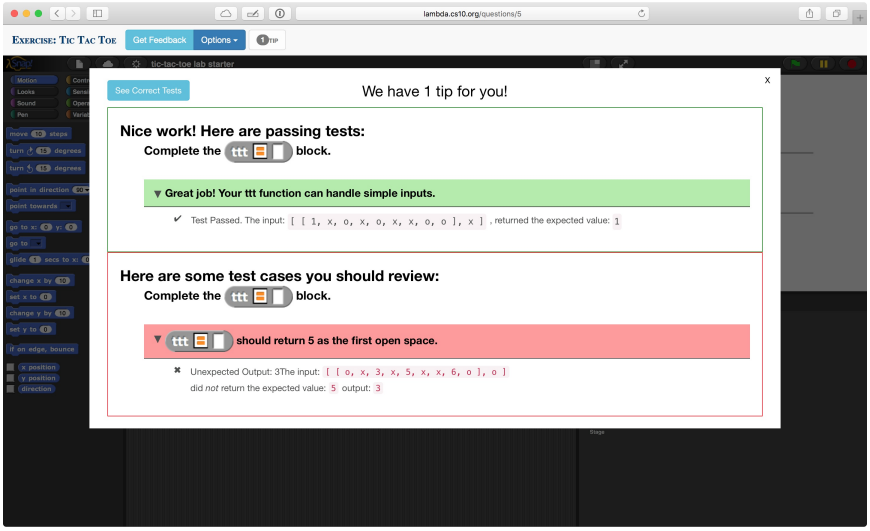

Figure 11: *An example of feedback showing some failing cases.*

The two screenshots in Figures 10 and 11 show two different versions of the autograder presenting feedback. Students see passing tests on top in green and failing tests on the bottom in red. Each section has details about the particular test cases, and the bold headings can contain tips or guidance for students.

8 Implementation

The λ web application is built using the Ruby on Rails framework [16], and makes use of many common web technologies. The Snap*!* interface is implemented purely in JavaScript. The initial purpose of the system is not to be a grade storage but to connect with an existing gradebook or Learning Management System (LMS) so that any grade results will be integrated with the rest of course data. We chose this route because, in our experience multiple sources of grades are prone to errors and delays. By designing a system which doesn't *need* to store grade data, we can make a lot of simplifications and focus on more important features.

8.1 Ruby on Rails Backend

λ exists as modern "Software as a service" application. It's designed to be hosted by a cloud services provider, using a webserver and a separate database server.

8.1.1 The Need for a Web Application

The JavaScript that powers the autograding works entirely client-side, meaning as long as you have the test files, there's no need for an internet connection. This path was initially chosen for three reasons:

- Snap*!* is client-side, and evaluating Snap*!* projects on a server would require a significant amount of work.
- edX provides a custom problem type called `JSInput` [17] which gave us a clear path for integration with edX.
- Developing an autograder and a web application required more resources than were available.

While the entirely client-side path was a good decision, we ran into a number of issues by relying on `JSInput` and trying to keep all features client-side.

8.1.1.1 Challenges with `JSInput`

edX's `JSInput` problem type provides a JavaScript API for sending scores to the edX platform. It allows us to build in a custom version of Snap*!* alongside the rest of the content in edX.

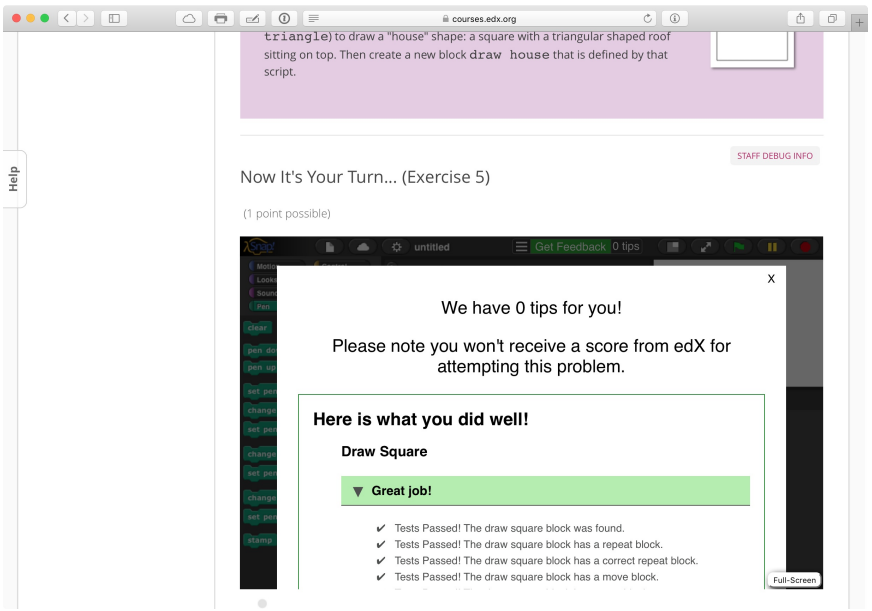

Figure 12: *Snap! can be embedded in edX through JSInput.*

However, we encountered several problems while developing the `JSInput` based integration:

- Developing code was very slow! Changes to code required manually uploading a new file to edX, which is a multi-step process. The libraries used for JSInput swallowed native JavaScript errors, making debugging nearly impossible.
- The edX interface has it's own mechanism for a "Check" button and showing feedback. Communicating the detailed output from the autograder didn't work very well, and we ended up developing many work arounds to get a seamless UI.
- There's no room for storing or retrieving user metadata. We rely on features that allow students to recall previous submissions. Through `JSInput` the only option for these features were to use the browser's `LocalStorage` API. This API has limits, like a max of 5MB of storage, that caused problems for some students.
- Furthermore, without a dedicated database, every single test file written had lots of hard-coded metadata that was repetitive and prone to error.
- While edX provides user logs for the entire course, we found dealing with these logs to be needlessly complex. They are slow to get, and the autograder results are difficult to separate from the rest of the course data. As such, we haven't analyzed the edX data to the extent we'd like to. A simpler logging system described below has been immensely helpful for our analysis.

The one benefit of these problems was that it forced the development of the autograder into two components: A JS interface to edX, and a "dumb" client-side component that sits on top of Snap*!*. This distinction was helpful when adapting the autograder to work with the new web application.

Perhaps most importantly: the grading system could only work with edX. CS10 uses Canvas [Citation not found] as it's LMS, and many high schools use different systems. The need to build a custom solution for every platform would be prohibitively expensive. Fortunately, the LTI protocol provides a decent solution for most of the tasks we'd like to accomplish.

At the end of the day, the decision to build an initial version tied to `JSInput` was a good one, as it was still probably faster than building a full web application at the same time.

8.1.2 Basic Architecture

λ is a Ruby on Rails (commonly abbreviated as "RoR") [16] web application backed by a PostgreSQL database. The database primarily contains a set of questions, a submissions log, and a users table, as well as some additional metadata. The current version is deployed to Heroku at lambda.cs10.org, but it could be deployed to any cloud provider.

8.1.2.1 Questions and Submissions

The core functionality is primarily supported by two, fairly simple, data models: `Question`s and `Submission`s.

A `Question` needs only three attributes:

- `title` : A human-readable ID for the question
- `points` : Points are used to normalize scores. (See the LTI section below.)
- `test file` : The test file is a JavaScript file (described below) which includes the test cases as well as feedback presented to the student.

Though they aren't currently used, future updates for will make use of the following properties:

- `content` : Currently, it is up to course staff to provide context for the questions which are being graded. In the future, the λ will display this content alongside the Snap*!* interface.
- `tags` : Questions can contain tags which can aid in searching, or trying to correlate student performance across problems. There is also potential for using tags to recommend problems to students as a study tool.

A `Submission` has a few key properties:

- `test results` is a JSON-formatted result from the autograder. It contains the points given to each test case as well as the specific results and feedback.
- `code submission` is a full export of the Snap*!* that students wrote.
- `user info` is a set of data about the submitting user which includes what source they came from (see the accounts section), and if they're a part of a course.

Note that logging submissions is purely for purposes of analysis and backup. By implementing the LTI protocol, the LMS will contain all necessary data for students to receive grades. However, if we choose to adapt λ to include resources for studying or question recommendation, the internal submissions database will become more important.

A `Course` is an object which manages the LTI connection, and needs only two values:

- `consumer_key` is a unique key for each course. This helps the application separate between different LTI consumers, primarily for purposes of analysis.
- `consumer_secret` is a hash of the `consumer_key` . It's automatically generated by the application when a Course object is created.

Note that the `Course`s table isn't entirely necessary. The LTI connection's `key` and `secret` values *could* be entirely static (i.e. in an environment configuration), but such an approach is prone to errors and has security concerns as an application is connected to multiple systems.

8.1.2.2 User Identities

When building a tool with grading data, it was critically important that we had an easy and way to identify students, and to minimize the need for an additional login.

8.1.2.2.1 LTI

The *IMS Global Learning Consortium* [18] is a standards body composed of educational instituions, intrest bodies and edtech companies. IMS publishes a specification called LTI [19] which can briefly be described as "OAuth for educational applications". The LTI authentication process is actually based on the OAuth [Citation not found] protocol, but it's designed to be completely seamless for students. (Unlike a Google or Facebook authorization, a student who is already authenticated inside a LMS does not need to specifically 'authorize' an application when LTI is used.)

The LTI protocol defines two "categories" of applications: a `Tool Consumer` (TC) and a `Tool Provider` (TP). λ is a provider, while the LMS is a typical consumer, in this case bCourses (Berkeley's instance of Instructure Canvas). A typical user flow involves a student visiting an assignment page (inside a LMS) which contains either an `iframe` element or a special link. Currently, λ implements version 1.1 [Citation not found] of the LTI specification, though support for version 2.0 [Citation not found] is planned. LTI 1.1 only allows a tool provider to read and send grades for a currently logged in student, so λ must work within this limitation. LTI 2.0 will potentially allow for more data about courses to be shared with students (such as handling parter assignments), but it so far not well supported among LMS vendors.

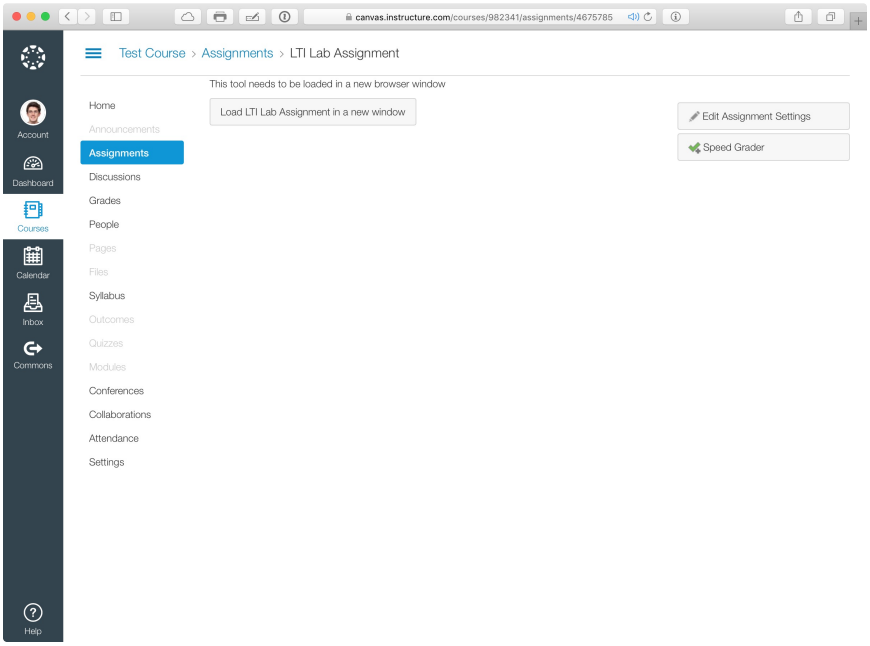

Figure 13: *When a student clicks on the link, a new tab will open with the proper question they are assigned. Clicking the "Get Feedback" button triggers a submission which sends the grade back to the LMS.*

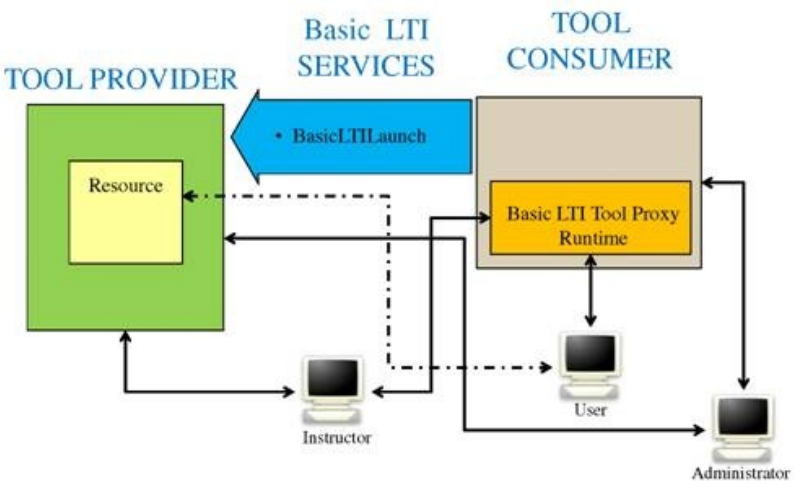

Figure 14: *A very basic LTI launch sequence. Image from the IMS [Citation not found].*

When a user clicks the link (or an embedded `iframe` is displayed), a HTTP `POST` request is make to the provider which includes application-level configuration data (including a `consumer_key` and `consumer_secret`). The challenge is that the current version of the LTI protocol requires that this `launch_url` be the same for every assignment. (In this case the URL is `https://lambda.cs10.org/lti/sessions`.) After completing the OAuth handshake, the TP (our application) checks for the presence of additional configuration info passed by the TC:

- If a `question_id` is provided, then λ will load the specific question. Currently LTI doesn't provide a standard interface for loading a specific resource, so if an instructor passes in this value, students will automatically be redirected to the proper question.
- A grade passback URL is optional. If the application sees this URL, it will post a score back to the TC. If the URL is missing, then λ skips posting a score but still saves the submission to the local database.
- User Info: If the instructional staff choose to, they can configure the LMS to send "Public" information to λ. (Public here means, what FERPA defines are directory information. In the case of UC Berkeley, this includes the student's name, email, and user ID, specifically different from their student ID. **REF?**) If the TC doesn't send public information, then it will send an obfuscated hash to identify each student. Again, this primarily only affects the analysis capabilities provided to TAs.

8.1.2.2.2 Need For (Regular) OAuth

Unfortunately, despite the advantages of LTI, we found that traditional OAuth was still a necessary component. The primary motivation was the need for site admins, and TAs who can login without having to go through a LMS. We're using Google as an OAuth provider in addition to LTI. In the future, we would also like to connect the Snap*!* cloud account system through OAuth, but this is waiting on enhancements to the Snap*!* cloud.

This feature can be useful for students as well, once we build out student dashboards. However, in order to be effective, we'll need a way to associate LTI accounts with OAuth accounts. Though not fully implemented, we will be able to automatically associate accounts for students if they share the same email address. For campuses setup like Berkeley, this will the default case for most students since their Google account and the LMS account originate from the same campus systems. If the emails don't automatically match, we should be able to provide this functionality by emailing activation codes.

8.1.3 Security Concerns

Finally, we need to discuss a significant concern about security. Currently, the autograder test files are implemented in plain JavaScript, and are served alongside the rest of the page content. This means there's a fairly gaping hole allowing for Cross-Site-Scripting (XSS) vulnerabilities. The current mitigating factor is that only trusted accounts with an admin flag can upload test cases. This is an acceptable limitation for now, but in the Future Work chapter we describe a potential way around this by allowing test cases to be written in Snap*!*, then securely complied to JS on the serverside.

8.2 The Autograder Interface

The autograder components are built in plain JavaScript and HTML. Great care has been taken to avoid modifying the Snap*!* environment as much as possible. The primary reason for this is to allow the autograder to be easily updated along with new versions of Snap*!*.

9 Experimental Setup

In the Spring 2016 semester, CS10 [9] used the autograder as part of the routine for lab checkoffs. In total, the autograder was used for three different labs, but students were only required to try the autograder for one lab.

9.1 Oral Lab Checkoffs

CS10 uses oral lab checkoffs as a means of granting credit for a lab. The way this typically works is that students are given one week from the assigned lab date to check in with a TA or lab assistant. Students are checked off when they successfully answer two or three questions about the lab, and usually are asked to present an example of working code.

It's worth mentioning that these checkoffs are designed more around completeness and effort placed in the lab, rather than absolute correctness. (Though, whether or not to grant credit is up to the individual TAs.) Oral lab checkoffs provide a good reason for TAs and students to interact more often. Over the past few semesters, the model of lab checkoffs has been refined based on student and TA feedback, and is *designed* to require only minimal additional effort for the course staff to manage, and for students complete. We're comparing the autograder against a fairly strict standard in this scenario, which we hope illustrates the potential.

9.1.1 Challenges of Oral Lab Checkoffs

Many of the challenges faced with oral lab checkoffs motivate the development of an autograder. Lab checkoffs can take anywhere from 2-10 minutes per student, or per pair of students. During a busy lab section, particularly close to deadlines, this time can create a large backlog in order to get students checked off. While the time discussing with individual students is valuable, the creation of a queue can waste student's time if they don't need additional help. Furthermore, if most of the lab work can be done at home, then oral checkoffs can be inflexible for students who have a harder time making it to lab, such as those with disabilities or families.

Finally, traditional "clipboard style" methods for dealing with inputting scores for oral checkoffs is a fairly complex and slow task, where grades can get lost. To get around this, we put significant effort into a semi-automated system which allows TAs and lab assistants to input grades quickly and securely. Without such an investment, oral lab checkoffs would be much less practical.

9.1 Autograder Checkoffs

In Spring 2016, there were a total of 18 labs, 14 of which used Snap*!* [20]. We had our online autograder setup for 3 of the later labs:

Lab #	Lab Topic	Autograder
Lab 11	Recursion Part 2	*Required*
	Question: Complete the definition of `merge sort`	
Lab 12	Tic-Tac-Toe	*Optional*
	Question: Complete the `ttt` solver block.	
Lab 14	Higher Order Functions	*Optional*
	Question: Complete the `is _ pandigital?` block	
	Question: Complete the `min value of _ over all numbers in _` block.	

Note: *optional* means that the students could receive credit for the lab checkoff through either the autograder or an oral lab checkoff. *Required* means that the only the autograder was accepted for credit.

9.2 Reasoning

We chose this setup for a variety of reasons, trying to balance a new technology with the need to test it out.

- One lab was required so that every student would try the autograder at least once and could provide feedback about their experiences.
- Labs 12 and 14 were optional because we didn't want to penalize students if they felt the autograder was more confusing or stressful, but we wanted to give as many people a chance to use it as possible.
- Finally, this hybrid model for checkoffs will likely be the basis for future semesters of CS10, as more questions and feedback are written.

9.3 Challenges of Autograder Checkoffs

Naturally, even the most advanced autograder will be no match for a TA...yet. We fully recognize (as do the students) that an autograder cannot replace the detailed, specialized and more articulate of feedback of a TA. Furthermore, compared to the current webpage for lab checkoff questions [20], even the easiest-to-use autograder will be significantly more work than posting a few sentences to a web page.

However, when deployed in the classroom with a "hybrid" model of both lab and oral checkoffs we think we can alleviate the in-class pressures of checkoffs but still give students credit for the work they are doing. By designing a model that gives students more choice, TAs will hopefully be able to give more time to the students that need it most. This still leaves the problem that an autograder is still more work than the status quo. We think that this will become less of a problem overtime as a database of questions is built up and if the analytical capabilities prove to be useful to teaching staff.

10 Results

Over the course of three lab assignments, we found that students used the autograder to complete labs outside of course time. While we have no basis for when (or how often) students completed labs at home without the autograder, this shows that students are at least receiving the benefits of feedback when a TA is not present.

Secondly, when we surveyed students, they were almost evenly split between preferring oral lab checkoffs or the autograder. Given the lab setup, and the differing advantages for each environment, this is what we would hope for.

Note, unlike other educational studies, we are *not* trying to claim any learning gains from this system.

10.1 Usage & Statistics

The first part of our analysis will be to look at how often and when students were submitting their labs. This data was obtained from aggregate information in the λ Submissions database as well as from bCourses to compare to non-autograded labs.

10.1.1 Basic Statistics

- Total Number of students in CS10: 169
- Total Number of student using the autograder: 145 (86%)

Lab #	Oral Checkoffs	Autograder Checkoffs
10	145	N/A
11	34	133
12	57	80
13	132	N/A
14	97	64
15	142	N/A

Note this data has some anomalies due to bug in the initial setup, and confusion among course staff:

- Lab 11 had a bug posting scores through LTI for the first day. Some TA's mistakenly submitted scores twice. There is some overlap of those 34 students.
- Lab 14 had a bug in one exercise causing many students to get checked off by both methods.

We can look at the number of times that students tried the autograder:

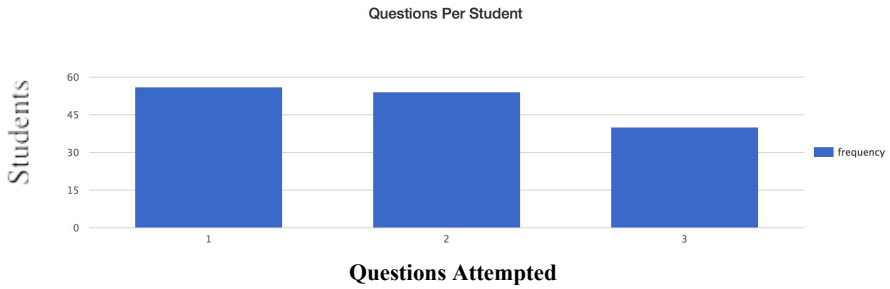

Figure 15: *Number of students by number of questions attempted*

What this shows us is that, though students were only required to complete one lab using the autograder, nearly 2/3 (93/149) found the autograder compelling enough to try a second time. From talking to students and staff, a portion of the drop off in students using the autograder may be due to the fact that the autograder doesn't handle pairs of students. Given that the later labs are some of the more difficult, many students may choose to work in pairs.

10.1.2 Submission Times

The second thing to look at is *when* students are submitting their work. While we certainly still want students to attend lab, improving the "at home" experience for students would be a significant benefit. Here, we see that students are choosing to use the autograder at home, with usage patterns that you would expect from undergraduates.

These charts show the overall submission times for each of the three labs. The red bar indicates the date that the lab was due for full credit.

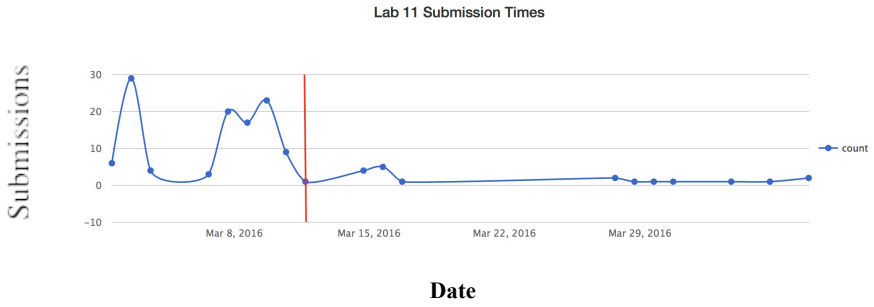

Figure 16: *Lab 11: submission times by day.*

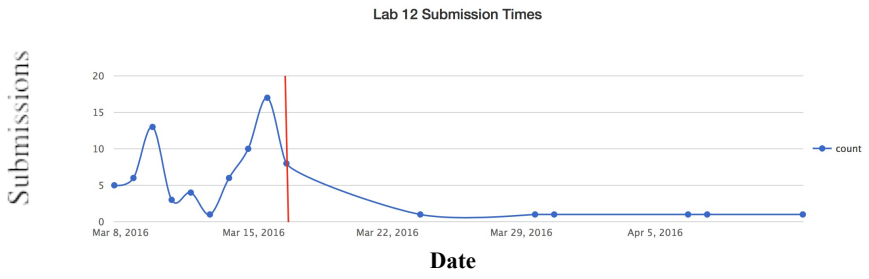

Figure 17: *Lab 12: submission times by day.*

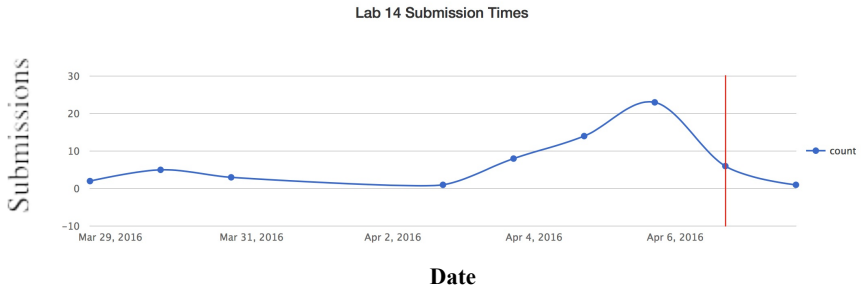

Figure 18: *Lab 14: submission times by day.*

These graphs don't reveal anything too surprising. For the most part, students are using the autograder to complete labs at the same pace they normally would.

We can look at the days of the week as well as the time of day to see when students are working on labs. Normal lab times are usually between 9:00-19:00, though this varies by the day of the week.

Figure 19: *Number of autograder submissions by hour of the day.*

Figure 20: *Number of autograder submissions by day of the week.*

Again, these graphs are fairly close to what course staff would expect. Most students are continuing to complete labs during their scheduled time, but a significant number are working at home.

10.1.3 Staff Time Savings

While time savings are not a primary motivation for this work, we should consider the potential benefits that could be saved be allowing students to get checked off using the autograder. Anecdotally, oral lab checkoffs take between 2 to 10 minutes to complete, with the average time somewhere around 3 to 4 minutes. The current usage patterns (as well as the results in the next section), suggest that we can expect 33%-50% of students to use the autograder.

If we have 15 labs which use Snap*!*, and 150 students completing labs, and assume that a lab checkoff takes 3 minutes: 33% of students the autograder would free up approximately 38 hours of TA time. (This is slightly under 30% of the total workload for a single 8 hour per week TA appointment at Berkeley.)

However, if we have 15 labs which use Snap*!*, and 300 students completing labs, and assume that a lab checkoff takes 4 minutes: 50% of students the autograder would free up approximately 150 hours of TA time. (This is 110% of the total workload for a single 8 hour per week TA appointment at Berkeley.)

Naturally, this might lead one to question the value of oral lab checkoffs. That's not the goal here at all. Oral lab checkoffs have been an incredible positive pedagogical tool. While writing tests can take some time, the hope is that the cost would aromatize itself well over many semesters.

10.1.4 Submission Patterns

We can also look at how often students attempt each question. This shows whether students are using the autograder more as a feedback tool, as a crutch or as simply a credit mechanism. While there's no clear exact number of times a student should use submit their work, it's clear that we want an overall 'happy-medium'.

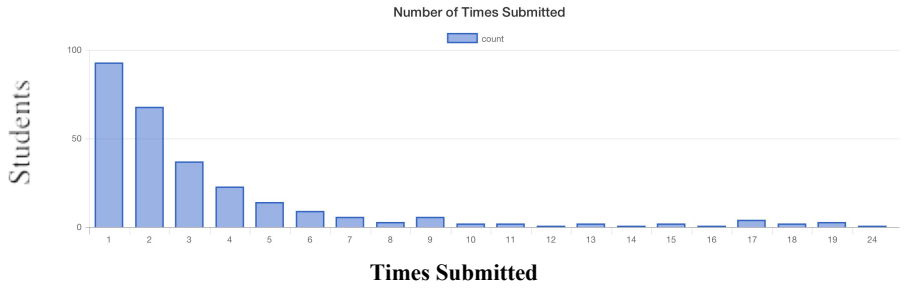

Figure 21: *Most students appear to only submit once at the end of their work.*

The data show that most students appear to be submitting only once, meaning they're not currently getting much benefit by the feedback presented. If there were more feedback presented, or potentially more challenging questions this might change. Though not yet implemented, non-graded feedback such as code quality suggestions might change the way students work to use the autograder.

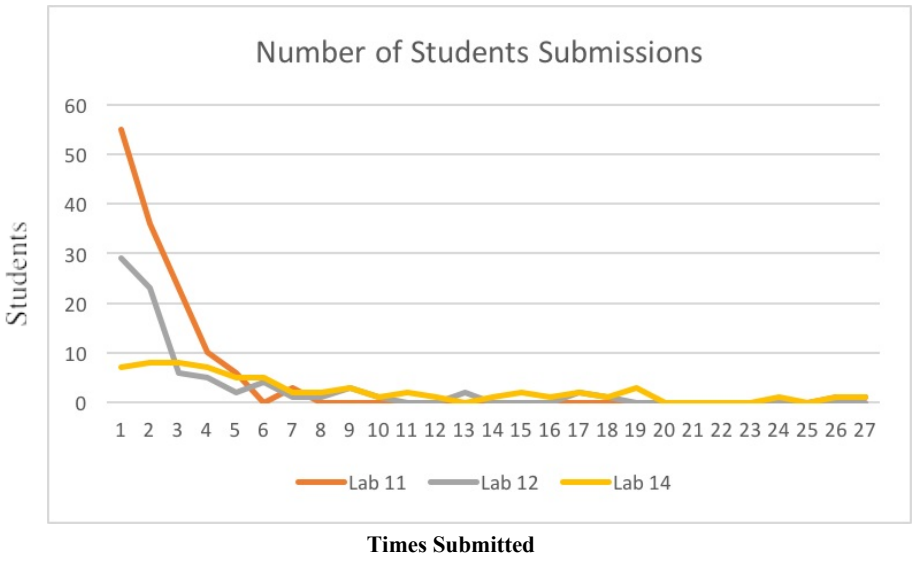

Figure 22: *Number of students by number of times submitted for each lab.*

These results also how a really long tail for the number of submissions by some students. This is to be expected with any autograder. However, from looking at the data, and from TA reports many of these high submission numbers may be due to the previously mentioned bugs. (Students tried submitting many times simply hoping that the errors would disappear.)

However, the difference in attempts for lab 14 is pretty clear. While most students still only submitted a few number of times, there is a much wider diversity in the number of questions. The number of blocks graded for lab 14 was 4 compared to the 1 or 2 for labs 11 and 12.

10.2 Survey Feedback & Analysis

Along with the data we collected, we surveyed students at two points to get feedback about their use of the autograder. The first survey occurred during the CS10 midterm, shortly after students should have completed labs 11 and 12. The second survey was given along with the final exam, at the end of the semester.

10.2.1 Midterm Feedback

The first question we asked students was whether they preferred online lab checkoffs or oral lab checkoffs.

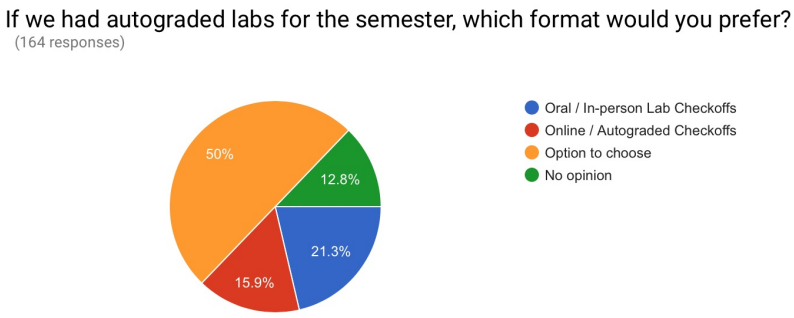

Figure 23: *Students are fairly evenly split between preferring online vs oral lab checkoffs.*

When discussing with TAs this is almost the exact opposite of the results they initially expected. TAs (and instructors, including from other courses) expected almost all students to prefer using the autograder to getting checked off orally. (Ratios some staff suggested were 75/25, 80/20, 90/10 in favor of an autograder. While no one had exactly the same prediction, everyone we spoke to assumed >= 70% students would prefer an autograder.) This validates that we have done a pretty good job in designing and tweaking the oral lab checkoff system (which has evolved over the past few year), but also that students like talking to TAs and that students are not simply trying to beat the system. Instructional staff had concerns about students preferences, because many assumed that the autograder would be the path of least resistance for students. It's worth noting, however that part of the bias towards in-person over online checkoffs could very likely be attributed to bugs in the system early on, but this is still a promising result. (Many open-ended feedback comments mentioned bugs or glitches as a reason for the feelings about the autograder.)

Here is a sampling of comments students gave. Each of these comments is representative of feelings of other students.

> "It would save a lot of time during check off."
>
> "You get to talk to an actual person! If any questions remain, they can easily be answered!"
>
> "If I don't finish in class, I am more incentivized to finish it at home instead of at next lab."
>
> “It worked for me, but I guess doing manual lab check-off was better, in a way, because you get feedback from the TA.”
>
> “A heads-up would have been nice, as well as something explaining how to use it, but I like this feature a lot. When it works properly, it is very useful and helpful for my learning.”
>
> “I don't trust it.”
>
> “The autograde sometimes has bugs. For example, if my code is wrong but sometimes still reported the correct value, it would mark me as a pass.”

Many students preferred the higher fidelity of in person question asking and answering. What’s interesting is that autograded checkoffs wouldn’t prevent students from getting their questions answered. Many students remarked about different levels of stress which come from automated systems or humans. While students didn’t elaborate on *why* they would be more (or less) stressed with an autograder system, the most probable answer is that such a view is really a matter of personal preference. Many students do enjoy that (human) TAs are much more relaxed and comfortable to discuss questions and generally forgiving of errors. In contrast, the autograder (currently) can only provide static responses, and is very brittle when determining correctness. However, some students also noted that they’d prefer not to talk to a TA "unless absolutely necessary" and that they found the automated format easier to deal with. We think this is yet another demonstration that a ‘one-size-fits-all’ model isn't usually the best approach.

When we asked for general feedback about the tool, most of the comments were that it was confusing to use. This is understandable, because in retrospect there was not enough documentation nor TA trailing before the autograder was launched in class. Fortunately, this is a fairly easy problem to address for future semesters.

10.2.2 Final Survey Feedback

One promising result, is that we asked students for feedback on the overall use of lab checkoffs. While there is (not surprisingly) a large contingent of students who dislike the lab checkoffs, most of the students in support of checkoffs specifically asked for more autograder questions.

When asked for feedback specific to the autograder, students seemed more positive than they did on the midterm survey. Of the students who had complaints, "bugs" and "glitches" were the biggest reasons. However, as with the midterm survey, there is a large number of students who are very strongly in favor of the opportunity to talk to TAs.

The final questions that we asked mostly backup the data that was collected.

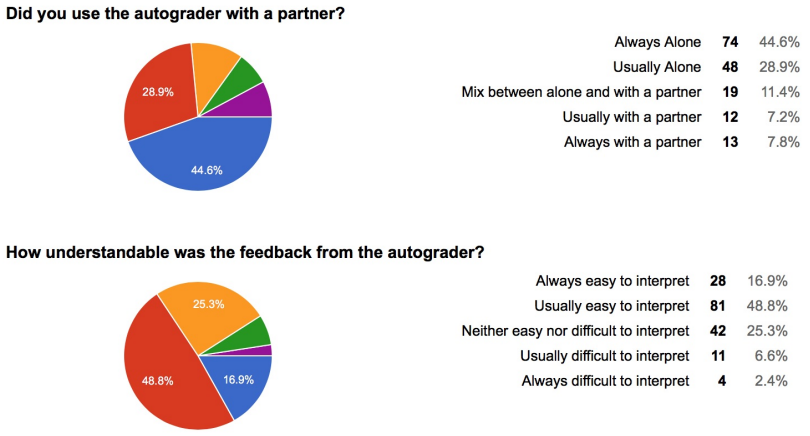

Figure 24: *Most students completed checkoffs alone, and found the feedback easy to interpret.*

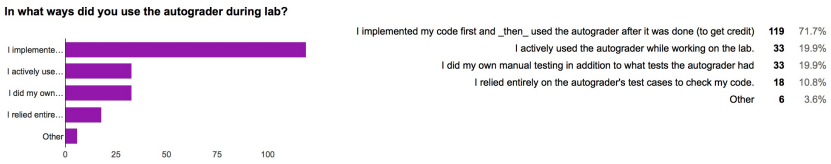

Figure 25: *Most students reported completing work before using the autograder*

So far these results confirm the data that's been collected. In the future, this suggests TAs can introduce the autograder to students as well as present the motivations for *how* students should think about using it as a study tool.

11 Future Work

Thus far, we have shown the current system has enough capability to be used in the classroom. However, we see a number of areas where λ could be expanded. Additionally, some of the analysis has revealed the need to more closely explore at some specific research questions.

11.1 Server-Side

The server-side improvements to λ are numerous, but we wanted to highlight a few that we think could provide the biggest benefit to students and instructors.

- Question Tags and recommendations.
  - λ's data model already includes question-level tags, but they aren't currently used for anything. We think it would be immensely helpful as a study tool if students could be recommended problems based on related tags for questions they have struggled with.
- Better (User-Friendly) Dashboards
  - Writing SQL queries to operatre the current dashboards is not a viable solution for the future. When a logged in as an administrator there should be links for each item (like a course or a question) for instructors to view basic statistics.
  - Support drilling down to individual test cases when analyzing questions. A well written autograder test covers many types of mistakes. TAs should get feedback about which particular tests students are struggling with the most so that they can address those concepts in class.
- Integrated display of question instructions.
  - The current version assumes that instructors will give students directions in some place other than the autograder. While this works OK, it would be much nicer if students could chose to hide or show instructions on demand.
  - This is actually a feature regression from edX, where the autograder was always embedded alongside the course content.

11.2 Autograding

Aside from continual bug fixes, there are four key areas for improving the autograding capabilities.

1. **Clearer, more explicit feedback**
   - The current output provided by the autograder is focused mostly on the results of individual test cases, by comparing the expected vs returned outputs of blocks. While test authors are able to work individually written feedback into thes test cases, the overall presentation isn't very clear. Furthermore, because there's not much room on the screen for good feedback, authors don't have an incentive to write as much.
2. **Snap*!* Test API improvements**
   - Currently, it can be fairly cumbersome to write tests in JavaScript. There is a lot of boilerplate code, and cleanly presenting images of blocks (instead of just a text version) requires a lot of effort for test authors.
   - One initial solution to this problem is the migrate the current object-oriented API to a version which is based more around configuration than code. While it couldn't be 100% configuration because authors will need to write custom test functions in some cases, a format which operates more like a JSON file would be easier to manage.
   - Snap*!* tests should be writable in Snap*!*. By using a Snap*!* feature called "codification" we are able to compile Snap*!* code to JavaScript (or any other language). Here is an example of a prototype Snap*!* library for writing tests. By compiling Snap*!* to

JavaScript we can also improve the security of tests that are written, because the set of functions available in Snap*!* is easier to restrict (and catch violations) than JavaScript.

3. **Static analysis capabilities**
    - We've included some very basic static analysis capabilities in the current version. However, they're tricky to use and require a lot of custom code. Through static analysis we should be able to give students targeted feedback about how to improve their code.
    - Some tools that would be useful are parttern matching functions for code structure (like finding all `if (condition) {return condition}` type structures).
4. **Image-based autograding**
    - We've explored image based autograding for problems like fractals and other turtle graphics exercises. Images would be an OK way to judge correctness, but they provide very limited feedback to students about how to improve.
    - Some paths for improving the feedback from image-based autograding would be to try to account for specific errors. For example, you could rotate two images until you find the minimum difference. If a rotation makes two images match, then you could provide feedback about where such an error might occur.
    - Other more advanced algorithms such as RANSAC could be explored.

11.3 Research Directions

Finally, now that we've shown students are interested in using an autograder, we should carry out more targeted research.

- What types of interventions help students learn or complete labs?
    - Instead of simply presenting feedback when students fail a test, we should have targeted interventions that ask questions or try a different method for solving the problem.
- Can we automatically generate better hints to give to students?
    - This is a highly active area of research, and now that λ is collection student data we could be able to use this to improve the feedback for future courses.
- How does the autograder affect motivation in early CS courses?
    - Some students are understandably concerned about the affects autograder have on their motivation to complete a task. While there is some existing work on this topic, we believe there's room to explore this in the context of visual programming languages.

12 Conclusion

We present λ, a system for autograding Snap*!* programs. The tool contains both a novel front-end student-centric autograder that presents immediate feedback to students, as well as a backend database that can serve as question repository. After successfully being used on edX, we've shown that the tools can be adopted in a traditional classroom. To accomplish we implemented the LTI protocol, so that λ may be used with as many LMSs are possible with the goal of reaching more students. Additionally, we've seen the benefits of having complete control of your infrastructure.

Our initial surveys have shown that while many students enjoy using an autograder, we must be careful not to replace instructional staff with technology. The stress that many students reported from the autograder suggest that there is a lot more human factors research that can be done about applying new technologies to the classroom. Still, we are confident that smart classroom policies which give students choice will continue to allow for the benefits of autograders without penalizing students who are uncomfortable with them. This does leave concern for online-only learning environments, but we see even a primitive autograder as a big step above nothing. Future interface enhancements and new features for studying will hopefully only add to the benefits students see while using the autograder.

Despite some successes, we know that there is a long way to go. We are still only able to grade a subset of all possible problems we could assign to students. There are many active research areas in techniques like automatic hint generation which we think will prove useful in the future. However, we also recognize that despite the promise, we need to spend to lower the barrier to entry for writing new content, both in terms of saving time and following pedagogical best practices. Fortunately, we think there is a clear path forward in this area.

Above all, we have shown that it is possible to build an autograder for a visual programming environment. Though these tools developed out of a need to scale teaching computer science, we believe that have the potential to greatly enhance traditional classrooms. As an area of active research we are excited to see the directions that autograding visual programming languages may take.

13 References

1.	Undergrad Computer Science Enrollments Rise for Fifth Straight Year — CRA Taulbee Report - GovAffairs, http://cra.org/govaffairs/blog/2013/03/taulbeereport/
2.	Cuny, Janice and NSF, nsf09534 Broadening Participation in Computing (BPC) | NSF - National Science Foundation, 2009. https://www.nsf.gov/publications/pub_summ.jsp?WT.z_pims_id=13510&ods_key=nsf09534
3.	Exploring Computer Science, http://www.exploringcs.org/
4.	Board, The College and Diez, Lien, AP Computer Science Principles - A New AP Course - Advances in AP® - The College Board | Advances in AP, https://advancesinap.collegeboard.org/stem/computer-science-principles
5.	Harvey, Brian and Garcia, Daniel and Development, Corporation Educational, BJC - Beauty and Joy of Computing, http://bjc.berkeley.edu/
6.	About Us | edX, https://www.edx.org/about-us
7.	Harvey, Brian and Mönig, Jens, Snap! (Build Your Own Blocks) 4.0, 2016. http://snap.berkeley.edu/
8.	Scratch - Imagine, Program, Share, https://scratch.mit.edu/
9.	UC Berkeley}, CS10 | Spring 2016, http://cs10.org/sp16/
10.	{Code.org Code Studio, https://studio.code.org/
11.	Fraser, Niel, Blockly | Google Developers, https://developers.google.com/blockly/
12.	Code.org, code-dot-org Github, 2016. https://github.com/code-dot-org/code-dot-org/tree/staging/apps
13.	Johnson, David E., ITCH, *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*, *ACM Press*, 2016. http://dl.acm.org/citation.cfm?id=2839509.2844600
14.	Soloway, Elliot and Guzdial, Mark and Hay, Kenneth E., Learner-centered design: the challenge for HCI in the 21st century, *ACM*, 1994. http://dl.acm.org/citation.cfm?id=174809.174813
15.	Exploring a Structured Definition for Learner-Centered Design, http://www.umich.edu/~icls/proceedings/pdf/Quintana2.pdf
16.	Getting Started with Rails — Ruby on Rails Guides, http://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark
17.	4.3. Custom JavaScript Applications — Open edX Developer's Guide documentation, http://edx.readthedocs.io/projects/edx-developer-guide/en/latest/extending_platform/javascript.html
18.	IMS Global Learning Consortium, https://www.imsglobal.org/
19.	Learning Tools Interoperability v1.1 Implementation Guide | IMS Global Learning Consortium, https://www.imsglobal.org/specs/ltiv1p1/implementation-guide
20.	Huang, Rachel and Kuphaldt, Adam, Lab Check-Off Questions, 2016. http://cs10.org/sp16/labquestions/

Appendix A: Obtaining the Source Code

The code for all projects in this report is available in various repositories on Github.

- cycomachead/lambda contains the main Rails web application.
- cycomachead/lambda-evaluator contains the JavaScript source that talks to Snap*!*.
- jmoenig/Snap--Build-Your-Own-Blocks contains the source for Snap*!*.
- cycomachead/thesis contains the source for this work.
- The CS10 organization contains info about the CS10 course.
- The beautjoy and bjc-edc organizations contains the curriculum used in edX and CS10.

The graphs in this report were generated primarily using the blazer gem for Rails, and this report was produced using the GitBook ebook tool. You can read it online.