Submitted in part fulfilment for the degree of BEng

Including mutation testing as part of a continuous integration workflow

Tim Waterson
Supervisor: Louis Rose

26/04/2016

11,944 words, as counted by the TextMate word counter. 30 pages.
The word count excludes code listings and the Table of Contents and Bibliography.
Both the word and page counts exclude all appendices.

Abstract

Many software projects are now equipped with automated test suites of varying levels of comprehensiveness. Mutation testing has the potential to be a powerful measure of test suite effectiveness, but has failed to gain widespread adoption, primarily because it is very time-consuming.

Continuous integration systems have been deployed widely in industry and are used to, amongst other tasks, run test suites and thereby determine whether or not a given build is deemed acceptable.

This project provides software intended to enable mutation testing to be run with each CI build, and to be a determinant of build success. The produced application speeds-up Mutiny (a mutation tool for Ruby) through parallelisation, and analyses its results to give a binary pass or fail outcome. It also produces various types of report, including remotely-hosted HTML reports ideally suited to cloud-based CI systems such as Travis CI.

Evaluation reveals that this approach is practical and usable with several real Ruby projects. A speed-up of around 60% was recorded, with the potential for an even bigger improvement after further work on the parallelisation algorithm.

Statement of Ethics

This project has no significant ethical implications.

Contents

1 Introduction

Maintaining a high level of software quality is, or at least should be, of paramount importance for most development teams. One way in which quality is ensured is through automated test suites. In order to be useful, these test suites must be as comprehensive as possible.

Mutation testing is a measure of test suite effectiveness, and involves making small changes to a program's source code, with the aim of determining whether or not the relevant test suite can detect that each altered version, or mutant, differs from the original. A mutation score, representing the percentage of mutants thus detected, quantifies the test suite's effectiveness.

Continuous integration (or CI) systems are usually integrated with a software project's version-control system and start a build after each commit or check-in. After the build completes, a report is generated and, if the build is deemed to have failed, the responsible developers can be alerted by email or other means. These systems provide the advantage of making it easy to see exactly when, where and by whom low-quality code is introduced into a system, making it possible to quickly improve or remove such code.

If run regularly, as part of each CI build, mutation testing has the potential to improve software reliability. As will be described in the Literature Review, it has, however, failed to achieve widespread industrial adoption, primarily because it is slow. Also, the percentage score produced from mutation testing does not obviously match-up with the binary pass or fail statuses of CI builds. This project's goal is to make a small contribution to addressing these two deficiencies, and thereby to provide software (which I've called MultiMutiny) that will enable mutation testing to be easily included as part of CI builds and to give it sufficient performance to be somewhat useful.

The Literature Review goes over the above definitions and areas of work in much greater detail, encompassing basic manual and automated testing techniques, before moving on to code coverage metrics. Mutation testing is introduced as a stronger version of code coverage. CI systems and their benefits are explained, after which there is discussion of some issues inherent in mutation testing and how those may hinder operation in a CI context.

The main purpose of the Analysis section is to give the rationale behind three key decisions relating to the design of this project's software; the decision to build a wrapper around existing mutation testing tooling, the choice of which tool to use for the initial implementation, and the decision (and its subsequent experimental validation) to exploit the fact that many test suites and mutation testing tools are single-threaded and use a parallelisation-based approach to speed-up the mutation scoring process. This section also provides justification for the creation of a cloud-based reporting component and for the decision to use a static, user-determined threshold to determine build success.

The Design & Implementation section describes the final structure and implementation of the client-side MultiMutiny application and its cloud reporting component, as well as the relationship between the two. This section also explores the way in which the main, client-side program is designed to be as extensible as possible. There are also some brief comments relating to the software development process followed while undertaking this project and a description of MultiMutiny's parallelisation algorithm.

The Evaluation section attempts to determine the extent to which the software produced as this project's main deliverable meets its goals and requirements. This is done by running MultiMutiny against several real codebases, showing that it works and that a significant speed increase is achieved through parallelisation. There is still room for improvement with regard to the latter point however.

Finally, the Conclusion provides a brief recap of the project and gives a few suggestions for further work.

2 Literature Review

The two main concepts that this project builds upon are mutation testing and continuous integration. Both of these are related to testing software, and thereby evaluating software quality, so this section begins with a brief overview of the basic principles of software testing, before moving on to automated software testing methods. Code coverage measurements are then explained, before mutation testing is introduced as a stronger measure of the quality of an automated test suite. A fairly brief overview of continuous integration, again in the context of automated testing, is then given before the narrative moves on to two of the main challenges associated with, to quote the project title, "including mutation testing as part of a continuous integration workflow", work in the literature related to addressing those challenges, and finally the goal and motivations of this project.

2.1 Manual and automated testing

For as long as there has been software, there has been an obvious need to ensure that that software does what it is supposed to do. In other words, to ensure that the software is of high enough quality, and that it meets the relevant requirements. The wider discipline of software testing has been the subject of innumerable papers and textbooks and is nowadays quite broadly defined in the literature, encompassing both finding bugs and the, in some cases quite subjective, evaluation of everything from program architecture to user interface accessibility and the comprehensiveness of documentation[1, p. 6]. Important as those wider measures of quality are, it is the traditional (and some say outdated[1, p. 5]) definition of testing, as finding bugs, that is relevant for this project.

The apocryphal original computer bug was a moth found in one of the relays of a Harvard University computer in the mid-1940s, and subsequently sticky-taped into the pages of a logbook to be preserved for all time[2]. Today however, the bugs found in programs are more likely to be of a less literal nature: They are implementation problems (i.e. incorrect code) within the software itself, that cause an incorrect output or a failure to output anything at all. To elaborate on the above more formally, and using the definitions given in [3, p. 12], the defect within the source code is known as a *fault*, an erroneous internal state arising from that fault is an *error* (which may or may not be corrected through, for instance, exception handling code), while an external manifestation of an error (such as the infamous "blue screen of death") is a *failure*. In this language, *testing* is the process of finding failures, while *debugging* is that of finding a fault given a failure.

The original, and least complex, method of testing is manual testing carried out by human testers. Sample inputs are given to a system and the output compared, by the tester, to that which is expected. Imagine a program, known as `caseflip` and referenced throughout this section, that transforms a single input character according to a simple rule: If the input character is lowercase it will be transformed to uppercase and vice-versa, otherwise it will be output as-is. Thus, `a` will become `A`, `Z` will become `z` and `6` and `+` will both remain unchanged. The aforementioned examples would actually make reasonably good manual test cases, at least in the absence of any knowledge of the system's internal mechanisms. Testing without knowledge of a program's source code is known as *black-box* testing, while testing with that knowledge, which can help tailor tests to the actual code-paths within the program, is *white-box* testing[4, p. 124]. It is the latter type of testing with which the remainder of this report is concerned, as it only with that type that mutation testing (and computing other test quality metrics) is possible.

```
def flip(char):
  codepoint = ord(char)

  if 64 < codepoint < 91:
    return chr(codepoint + 32)

  if 96 < codepoint < 123:
    return chr(codepoint - 32)

  return char
```

Code Listing 1: The important part of the source code for the `caseflip` application. A fuller code listing, which includes "boilerplate" code omitted here, can be found at https://github.com/timw6n/caseflip. The "magic numbers" in the `if` conditions represent the ASCII codepoints of various characters (see Appendix 1 for more details).

Straight away, some of the problems inherent in manual testing become clear: Relying on people to sit there performing tasks and checking output is expensive, in terms of time and thereby money. Books (such as [1]) written prior to the widespread adoption of automated testing frequently state that over half of the total cost in a software project is spent on testing[1, p. 3]. Moreover, the fact that manual testing is time-consuming both limits the number of tests that can be run to validate a change and tends to mean that testing is confined to a set period immediately prior to a release, rather than a full set of tests being run after smaller milestones. Manual testing is also error-prone, in that human testers may well miss subtle failures encountered in the course of a long and repetitive process. Monitoring a stream of output for errors, a common task during testing, is not entirely dissimilar to, for instance, monitoring CCTV monitors for untoward movements, and, for the latter task, there is literature pointing to a "vigilance decrement" after 20-30 minutes of work, after which the detection of something amiss becomes less likely[5].

Thankfully, quickly and repeatedly performing simple tasks is something computers are rather better at and, as one might expect, there have long existed a variety of tools to automate parts of the testing workflow. One of the most well-known unit testing frameworks (unit testing here being defined in the broadest possible sense, encompassing a lot of what might strictly be termed integration testing) is the open-source JUnit framework for Java. The code sample below shows unit tests for the example application introduced above, written using the `unittest` module in Python's standard library. This module was itself inspired by JUnit and has many similar features[6]. In systems of this type, a *test case* (such as `testLowercaseA` below) is an individual method containing (if it is to be useful) one or more assertions, while a *test suite* is the full set of test cases relevant to a program[4, p. 124]. Test suites can be sub-divided into one or more *test classes*, which usually mirror a similarly-named class in the main program. `TestFlippingCharacters` in the example below is a test class. A framework also provides for setup and tear-down methods to ensure isolation between different test cases and classes, as well as reporting systems to catch errors and provide statistics[7, p. 10].

```
class TestFlippingCharacters(unittest.TestCase):
    def testLowercaseA(self):
        self.assertEqual(flip('a'), 'A')

    def testLowercaseZ(self):
        self.assertEqual(flip('z'), 'Z')

    def testNumberOne(self):
        self.assertEqual(flip('1'), '1')

    def testUppercaseZ(self):
        self.assertEqual(flip('Z'), 'z')

    def testUppercaseA(self):
        self.assertEqual(flip('A'), 'a')

    def testPlusSymbol(self):
        self.assertEqual(flip('+'), '+')
```

Code Listing 2: Example unit tests for the method defined in Listing 1. Again, the full source file can be found at https://github.com/timw6n/caseflip.

Over in the Ruby world, which will be more relevant for the main part of this report, the most frequently-used testing framework seems to be RSpec. A set of example RSpec tests are given as the second part of Appendix 2, and, as can be seen, while the structure and keywords used are somewhat different from JUnit-type systems, what is accomplished is more-or-less the same. The "spec" in the name is short for specification, reflecting the fact that the "specs", or tests, for a class or method are often written prior to the code they test and serve as an automatically-checkable description of the required functionality[8, pp. 3–6]. This methodology is referred to as test-driven development, but, for the purposes of this report, the point when the tests were written, or the terminology used to describe them, matters little.

One final point to note regarding our example application is that this rather trivial program *can* be tested exhaustively. It would be straightforward to write a set of 128 tests (one for every ASCII character) and be confident that the resulting test suite verifies every possible program output. No further evaluation of the test suite's quality would be necessary. However, exhaustive testing is often impractical, if not impossible, for most real programs. The range of potential input combinations for a simulator for the board game Go, for

instance, greatly exceeds the number of atoms in the universe[9] while there are many programs, particularly those accepting arbitrary user inputs, for which the number of potential inputs are effectively infinite.

If one accepts that truly exhaustive testing is not possible for a program, then it follows that there is no set of tests that can guarantee to find every fault[4, p. 128]. That is not to say that there are no tools available to estimate the quality of a given test suite, and that such heuristic measures are not useful in practice, just that they cannot be taken as absolute. Two such test-quality scoring measures, namely code coverage and mutation testing scores will now be introduced.

2.2 Code coverage

Code coverage metrics are one of the most widely used indicators of the comprehensiveness of the tests for a given codebase. To simplify slightly, a given chunk of code is covered if, at some point during a test run, it is executed. There are several different levels of code coverage defined in the literature, with the most widely supported in commercial tooling being *statement coverage*, or C_0[10]. As implied by its abbreviation, this is the weakest code coverage metric: Several other, stronger but consequently more difficult to implement and to satisfy, coverage criterions also exist[4, p. 124] but need not be defined in this report.

C_0 determines whether each logical statement is executed[4, p. 137] and is sometimes, slightly misleadingly, referred to as *line coverage*[11] since the usual practice for most languages is to have one statement per line. Returning to the example `caseflip` application and tests, the final `return char` line is not covered using this criterion, as each character used in the tests will trigger one of the two `if` blocks. A test using a character with a codepoint of 123 or higher (such as `|`) would be needed for full coverage.

In a practical context, code coverage tools are usually used to produce a percentage score, which represents the proportion of statements covered. This can be used as a measure of a test suite's quality, since uncovered code is effectively untested. Some organisations also set a minimum score as a hurdle that must be passed before a version can be released, though, as will be expanded upon later, this practice can be problematic.

2.3 Mutation testing

Mutation testing, one of the aforementioned two pillars upon which this project is built, was first proposed in 1971 and the first related paper was published six years later[12]. Over the following four decades, the field has attracted continued and increasing academic interest: In [13], a survey of papers published between 1977 and 2009, 393 mutation testing-related papers were identified, 51 of which were published in 2009. One of the more prolific authors on the subject is Jeff Offutt, and indeed his papers form a large part of this project's bibliography.

One way to explain mutation testing is as a fault-injection technique for software[14, p. 1], the virtual equivalent of techniques intended to test the resilience of computer systems to hardware faults. Instead of injecting faults into a running system through man-in-the-middle devices on input and output pins, or through bombarding CPUs with heavy-ion radiation[15], mutation testing modifies the program-under-test's source code prior to runtime. This is done by applying a number of *mutation operators* to the program's code, each of which makes a single change, injecting a single fault and thereby creating a *mutant*[14].

These mutation operators, conventionally referred to by three-letter abbreviations, range from the generally-applicable to the highly language-specific: The ROR, or Relational Operator Replacement, operator substitutes relational operators for either a closely-related operator (e.g. `<` to `<=` as shown in Listing 3 below) or for the opposite operator (e.g. `==` to `!=`)[16] and is implemented across a large number of mutation testing tools, for a variety of languages[16]. At the other end of the spectrum are highly language-dependent operators such as the SIR operator targeting Python programs, which removes one element from a collection slice operator (e.g. `collection[3:8:2]` to `collection[:8:2]`)[16].

```
def flip(char):
  codepoint = ord(char)

  if 64 < codepoint <= 91:
    return chr(codepoint + 32)

  if 96 < codepoint < 123:
    return chr(codepoint - 32)

  return char
```

Code Listing 3: One possible mutated version of Listing 1. Note the `<=` in line 4, column 27.

The next step in the mutation testing process is to *score* the mutants, that is to execute the application's tests against each mutant. In the context of this project, this means running the relevant automated test suite. Those mutants for which a test failed (i.e. those that the test suite could distinguish from the original program) are referred to as having been *killed*, while mutants for which all tests passed have *survived*[14]. Sometimes there are also *invalid*, *incompetent* or *stillborn* mutants that won't compile (in the context of a static language such as Java) or fail immediately at runtime (in the case of dynamic languages such as Python or Ruby), usually because of a type violation[16]. These are counted within the killed group, despite not imparting any information with regard to test suite quality.

The main metric gleaned from the mutation testing process is the *mutation score* or *mutation adequacy score*[17]. This is usually defined as the number of killed mutants over the total number of mutants created[17], and is generally expressed as a percentage. Another definition that is sometimes seen in the literature, particularly in Offutt's papers, including [18] and [14], is of the number of killed mutants over the total number of **non-equivalent** mutants. An equivalent mutant is a mutant that functions identically in all respects to the original program[18] and so cannot be killed by even a perfect test suite. As a trivial example, the logical statements `a && b` and `!(!a || !b)` are equivalent. For reasons that will be explained later in this report, the latter definition is somewhat impractical in creating automated mutation testing tools.

Figure 1: The Mariner I rocket taking off[19]. Less than five minutes later the rocket had to be destroyed after a crash became inevitable[20]. The cause was a missing hyphen in the code for the guidance program[20] — just the sort of tiny error mutation testing techniques could potentially catch.

Another way to think of mutation testing is as a stronger form of code coverage. Recall from above that a given line of code is covered merely by being executed by a test. The fact that a line has mutation coverage provides a stronger assurance; information as to whether the tests actually verify the contents of that line.

A more formal argument sometimes seen is that mutation testing *subsumes* code coverage testing[3, p. 174]. An intuitive proof for this is quite convincing: Assuming that the mutation testing tool is sufficiently capable so as not to produce invalid or incompetent mutants, for a mutation on a line to affect the test outcome, that line must have been executed, otherwise its contents would be irrelevant. However, the devil is in the detail, in that this subsumption relationship is dependent on precisely which mutation operators are used — if a mutation tool has no operators capable of mutating the statements on a given line then it cannot impart any information with regard to that line.

A more convincing argument favouring mutation testing over code coverage comes from empirical research in [21] and [22]. These two papers found evidence that, respectively, a high code coverage score is not strongly correlated with test suite effectiveness, whereas a high mutation score is.

The above point regarding the assurances that mutation coverage can provide gives us the main reason why mutation testing is useful in real-world scenarios. If the program's tests cannot differentiate between the original code and the mutated code, then there is no assurance as to which version encapsulates the correct logic: A surviving mutant could represent the correct version of the program. Additional justification for this comes from two widely accepted software testing hypotheses; the competent programmer hypothesis and the coupling hypothesis. The former states that a hypothetical competent programmer will tend to write code that reasonably closely resembles an equally hypothetical "correct" version of the program in question[14]. In other words, the actual and ideal versions of a program will differ only by a few faults. Thus, the faults injected by the mutation testing process will be similar to those inadvertently added by real coders.

The coupling hypothesis states that a test suite sufficiently comprehensive as to detect all simple faults in a program will thereby also detect more complex faults[14]. More succinctly, complex faults are *coupled* to simple faults. In the context of mutation testing, this provides another point of justification for operators only making a single, simple change. It also explains why higher-order mutants, those created by mutating mutants[23], are rarely used and mostly considered unnecessary.

2.4 Continuous integration

The second main theme for this project is continuous integration or CI. This section will be somewhat shorter than that relating to mutation testing, both because CI systems are simpler and easier to understand and because, while important for commercial software companies, the topic has attracted less academic interest. A literature review conducted by the authors of [24] in 2013 discovered only 76 CI-related papers, far fewer than were found in the similar exercise for mutation testing mentioned above.

To backtrack a little, understanding the environment in which CI practices first arose requires one to imagine a stereotypical "waterfall" development process, in which programming stops and the software is deemed feature-complete before it is first built and all the components fitted together, or integrated. Only then does testing commence[25]. The practice of continuous integration, on the other hand, requires the integration and testing process be carried out continuously (hence the name) alongside development[25]. The need to run builds and tests so frequently implicitly requires that the process be automated, as does the original definition of continuous integration as one of the principles of the "extreme programming" methodology[26]. The benefits of such an approach with regard to software quality are clear: Errors causing compilation or test failures can hopefully be found immediately after entering the system, and the relevant developer and section of code identified, thus making the debugging swifter and easier[26]. Implementing CI also opens the door to continuous deployment, whereby features are deployed to production individually and almost immediately, rather than waiting for large and infrequent releases[26].

In practice, CI is usually implemented by means of a CI system or tool: There exists a healthy ecosystem of such tools, some designed to be self-hosted on a server within an organisation's firewall (including the open-source CruiseControl[27] and Jenkins[28] applications or Atlassian's commercial, closed-source Bamboo[29] product) and more modern cloud-based systems such as Travis CI[30]. That said, it is perfectly possible to bypass these tools and implement a very simple system based on the event hooks provided by version-control systems. These systems are all centralised to a certain extent, in that there is not a separate instance run on each developer's individual workstation but one per team or per project. Using a single, well-defined environment helps avoid a whole category of issues relating to build configurations, dependency and IDE versions and the like which may differ between individual developers. In the context of integrating mutation testing, it would also allow a single knowledgable developer to configure a mutation testing tool for their whole team.

The usual CI workflow is that, when a change is submitted to the project's version control system (i.e. a commit is pushed to a Git repository, or a changelist submitted to the Perforce mainline) a CI build, that is an integration and testing run, will be immediately triggered[24]. A small minority of organisations do prefer to run builds at specific times, perhaps at hourly or daily intervals[24]. The build will be uniquely identified by a sequential build number, and then will either pass (if all of the activities run as part of that build report success) or fail if at least one activity reports that the version of the project submitted is in some way unsatisfactory. A failing build will result in the team being notified by means such as email or RSS[25], or even lava lamp as per the figure below. The expectation is that "fixing the build", that is making changes so as to trigger a new build that will succeed, will be a high priority task[26]. Regardless of the build outcome, a report is usually produced

giving details of the activities performed (usually in terms of the command-line output from the build script), details of the code changes that triggered the build, and further build-related statistics[25].

Figure 2: One way of representing the current CI build status[31]. When the most recent build has passed the green lamp will be illuminated, otherwise the red lamp will be lit and the development team will panic.

A key characteristic of an effective CI workflow seems to be that builds are completed, and their outcome reported, within a reasonably short timeframe. To be of value, builds need to complete quickly enough to give feedback to the relevant programmer while the changes that triggered the build are still at the forefront of their mind[24]. [32] reports that, as build times increase, not only does the feedback provided by CI become less useful, developers begin to feel less inclined to frequently push code to the repository feeding the CI system, thus negating some of the benefits that CI brings. [33] gives a rule-of-thumb whereby builds should be kept to less than ten minutes for reasons very similar to those detailed above.

2.5 Combining mutation testing and continuous integration

In an ideal world, having now introduced both mutation testing and continuous integration, the logical next step would be to explore pre-existing papers in the literature that bring the two together. However, as might be expected given the previously-mentioned lack of academic interest in CI, I have been unable to locate any such papers. That said, code coverage scores are somewhat similar to mutation scores, and some interesting work has been done regarding their practical applications. In [11], the major pitfall identified is that coverage metrics are quite easy to manipulate (the example given is that the positioning of the braces surrounding blocks can influence the coverage score) and that, combined with the human tendency to optimise work to the criteria by which it will be judged, can lead to rather perverse outcomes. It is important to remember however that, as mentioned above, code coverage is a weaker metric than mutation testing.

More promisingly, one of the main weaknesses inherent in mutation testing, and consequently one of the main problems that a great deal of academic effort has been expended in attempting to address, dovetails quite neatly with one of the aforementioned key characteristics of tasks run with every CI build: This is that mutation testing is a time-consuming process, whereas the utility of a CI build diminishes as the time it takes increases. Indeed, it has been acknowledged that the slowness of mutation testing has been one of the reasons behind its failure to gain widespread industrial adoption[18].

Intuitively, the reason why mutation testing is time-intensive is immediately obvious: In the worst-case scenario in which all mutants survive (i.e. a score of 0% is returned), a naive mutation tool must execute every test case (which will pass) against each mutant individually. This means that the worst-case time complexity is the product of the number of mutants (denoted as m) and the number of test cases (k), so $O(mk)$. The received wisdom in much of the literature seems to be that m scales with the square of the number of lines of code in the program-under-test (n), making the overall time $O(n^2k)$[12]. Certainly, a large number of mutants can be generated for even relatively trivial programs: Coincidentally, both `caseflip` and the equally diminutive demo application given in Appendix 2 produce 19 mutants, while Offutt cites a Fortran program that produces 111 mutants from only 5 statements[14].

[18] identifies three broad areas into which work to speed up mutation testing can be categorised, namely "do fewer", "do smarter" and "do faster". Work within the first of these seeks to create and score fewer mutants without compromising the usefulness of the results produced too dramatically. Techniques within this category include *selective mutation* (choosing the smallest set of mutation operators that will reveal sufficient gaps in testing) and mutant sampling, which involves randomly deciding which subset of the generated mutants should be scored. "Do smarter" approaches include spreading the scoring workload across multiple machines, as discussed in more depth below, and also so-called *weak mutation* techniques, that seek to decide whether or

not a given mutant has been killed based on the program-under-test's internal state after reaching the relevant line, rather than waiting for that program to finish normally. Finally, "do faster" approaches include several novel techniques intended to reduce the time taken for each mutant to compile.

One of the key "do smarter" methods is parallelisation of the mutation scoring process, that is the running of the test suite against each mutant. By definition, the scoring of each individual mutant should run independently of any others, meaning there is little to no synchronisation required between the parallel tasks and so, theoretically, a linear speedup could be possible. Moreover, both mutation tools and testing frameworks are often themselves single-threaded, meaning that, as will be expanded upon later in this report, such an approach wouldn't necessarily require co-operation from the entire toolchain. [14], published in 1992, used what was then quite a novel pseudo-distributed system architecture to explore the performance increases gained by parallelising a Fortran mutation testing tool. As expected, an almost-linear speed increase was found. These results were repeated by a number of other studies[12]. Such multiple-core architectures are now commonplace, especially in the workstation- and server-class machines that might be used for CI purposes.

The other major challenge when running mutation testing as part of a CI build is that mutation tools produce a percentage score, while CI simply requires a pass or fail output. The obvious solution to that problem is to treat any mutation score less than 100% as a failure, but this approach is fatally flawed because of the impossibility of automatically and accurately detecting all equivalent mutants: A perfect, and perfectly-tested, codebase could still fall below perfect mutation coverage if an equivalent mutant is produced. While there is work ongoing to try and heuristically identify some equivalent mutants using methods including comparing programs after applying compiler optimisations[34], the problem is, in the general case, undecidable[35].

On a more positive note, there is an offhand mention in [18] of using a mutation threshold (the minimum mutation score required for a successful outcome) of less than 100%, though that paper gives no real indication of what such a threshold might be. Some inspiration can be taken from the world of code coverage, where 85% is apparently a common threshold used by a lot of companies[11] but the rationale behind that number is unclear to say the least. The popular Coveralls[36] cloud-based code coverage system uses three colours to denote coverage levels: Scores below 80% are red, between 80 and 90% are amber and a score of 90% or greater gives a green colour. That service also provides the ability to set a requirement that code coverage should not decrease between builds, which could also be a potential option over in the mutation testing world.

2.6 Project motivation

Finally, a few words of justification for the overall goal of this project: A relevant Offutt quote is that "all respectable software engineering research should have the eventual goal of helping real programmers build better software"[12]. This sums up the aim of this project, which is to make a small contribution to enabling the use of mutation testing in the kinds of CI workflows commonly in use in industry and academia.

3 Analysis

Recall that the aim of this project is to provide a means by which mutation testing can be included as part of CI builds. To achieve that objective, the software produced needs to include a mechanism for speeding up the mutation testing process, as well as providing a binary pass/fail output, not to mention the tooling necessary to actually perform the mutation testing process and to create and view reports from a CI context.

This chapter describes broad goals and requirements for the work carried out as part of this project, as well as the justifications behind them, and the methods by which those requirements were arrived at. The Design & Implementation section immediately following goes into the detail of how the software was implemented in order to meet the specified requirements. In terms of the actual sequence in which work was carried out, the activities described in the aforementioned two sections were intermeshed: As will become particularly clear as some of the experimentation done to justify the speed-up approach chosen is explained, the design described in the following section was arrived at iteratively and alongside some of the more detailed analysis work below.

3.1 Constraints imposed by the CI environment

First, it was necessary to consider some of the technical constraints associated with running within a CI environment. For the sake of brevity and because cloud-based CI systems tend to be the most restrictive, the Travis CI documentation will be cited below for elements common to most similar systems. Competitor products will be mentioned only when they differ in functionality. It is also important to note here that, even if only for testing while under development, the software produced as part of this project will also need to be able to function outside of a CI environment.

A CI build consists of a setup phase, used to install dependencies and set up the environment, and which it is unnecessary to consider further, and a build phase, which runs the actual build and tests[37]. That phase consists of a number of commands, executed in sequence. If every command returns an exit code of 0 then the build will be marked as having succeeded. If any command returns a non-zero exit code then the build will continue but, at the end, will be marked as failed[37]. Travis builds take place on virtual machines under that company's control, which are instantiated anew prior to each build and then torn down (and the resources used by a different customer) afterwards[38]. This means that one cannot simply rely on a user being able to browse the machine's file system or run additional commands to find out the precise details of the mutants created. It would be possible to output the mutant's source code to the command-line, which is persisted as the build log, but for a large number of mutants that would create a ridiculously unwieldy amount of text. The solution therefore is to transmit the mutant data to a separate system for safe-keeping and eventual viewing, leading to the requirement that the software created as part of this project have a cloud-based component with that functionality. Builds using Jenkins and Bamboo can suffer from similar constraints, although for those systems the testing VMs involved would at least be under the direct control of the organisation running the tests. Many such organisations would be reluctant however to keep machines running just in case a developer needed more details about a single build.

3.2 Decision on whether or not to extend an existing tool

A key decision that needed to be taken was whether to improve an existing mutation testing tool to give it the features required or whether to build something at a higher level of abstraction, in other words a wrapper around a mutation testing tool. The main reason behind my choice for the latter approach was that the work needed to integrate with CI systems is quite generally applicable: Most mutation testing tools, regardless of which language they target, still produce mutants, score them and produce a percentage score at the end of the day. Choosing to extend a single tool would limit the potential impact of this project unnecessarily. Choosing to build a wrapper also means that, as will become relevant in the next chapter, the choice of implementation language for the wrapper is de-coupled from that used for the tool.

3.3 Decision on speed-up approach

The other main aim of the software produced as part of this project was of course to speed-up the mutation testing process somewhat. The decision to pursue a relatively tool-independent approach ruled out many of the possibilities identified in the Literature Review: Selective mutation and weak mutation would require too much co-operation from, and development of, the mutation tool to be appropriate. Mutant sampling

techniques wouldn't necessarily require such deep integration, but the random nature of most such approaches would jeopardise the repeatability of and consistency between runs necessary for trustworthy CI builds. "Do faster" approaches involving tweaked mutant compilation would be too language-specific, and are inapplicable to dynamic languages such as Ruby (chosen as the initial target language, as described below) anyway.

That leaves parallelisation, a "do smarter" approach, as the main contender. Unlike some of the aforementioned approaches, this method would not only not require deep co-operation from the mutation tool, it would also mean that the exact same mutants would be end up being executed against the same tests: The results should, in an ideal world, be indistinguishable from those arising from the (single-threaded) tool directly. The fact that, as discussed previously, there are papers in the literature reporting quite dramatic (i.e. up to linear as more parallel processes are added) speed increases from this method was also a factor in its favour.

3.4 Choice of mutation tool to wrap

Having decided to go with a wrapper approach, the decision then arose as to which mutation testing tool to actually wrap. While the eventual goal would be support multiple tools, the choice of initial tool would have an impact on the wrapper-to-tool interface, and, given the time constraints on the project, I'd need to be able to get to a working implementation sooner rather than later in order to validate other aspects of the design.

In collaboration with my supervisor, it was decided to choose a tool targeting Ruby-language projects as my supervisor was familiar with the area and I was reasonably familiar with the language at least. Only two truly viable options therefore emerged, namely Mutant[39] and Mutiny[40]. To assist in choosing between them, and to govern some aspects of the wider design, I devised the following set of requirements as a kind of contract that the chosen tool must meet, or at least be modified to meet:

1. The frontend and backend of the tool (i.e. mutation creation and scoring respectively) need to be callable separately.

This requirement is crucial, in that, in order for the wrapper to distribute the scoring work between processes, it needs to be able to sit between the mutant generation and mutant scoring parts of the tool (as shown in the diagram in the next chapter). Mutiny already met this requirement through its `mutate` and `score` subcommands. Mutant had no such ability.

2. The tool needs to be able to provide the set of generated mutants.

As alluded to above, Mutiny was able to meet this requirement as its `mutate` subcommand saves all generated mutants to a `.mutants` directory under the current path. Mutant only provides its mutants at the end of the complete testing process.

3. The tool would ideally provide metadata relating to mutant generation.

This metadata could include the mutation operator used and the line and column to which it was applied. As shown by the word "ideally" this requirement was not as important as the others. Mutant provides this information in its command-line output only for surviving mutants. Mutiny did not initially have this feature, but it was added during the course of this project.

4. The tool needs to be able to receive a subset of mutants and then score only those mutants.

This is the major piece both tools lacked initially. I implemented this myself in a forked version of Mutiny, which was then superseded by a subsequent release in the upstream repository.

5. The tool needs to output mutation scoring results somehow, ideally in an easily parseable format.

Both tools meet this requirement by printing their results to the command-line in a manner that can be parsed with a regex. Obviously, support for an actual data interchange would be even better.

6. It needs to be possible to run multiple instances of the tool in parallel on the same machine.

Both tools would seem to be able to meet this requirement. Mutant already supports parallelisation through its `-j` option, which may actually be something of a negative: Difficulties could arise if attempting to override its in-built behaviour, particularly if there was time to implement multiple-machine parallelisation.

7. Ideally, the tool would be able to do a basic sanity check to determine whether or not it supports mutating a given project.

Mutiny supports this explicitly via the `check` subcommand. Mutant, on the other hand, does not have such an option but will quickly exit if it cannot find anything to do.

The notes above should give a clear indication that using Mutiny was already emerging as the preferred option. The pragmatic factor that sealed the deal was that Mutiny's sole developer was my supervisor for this project, meaning that help with understanding its workings would be easily accessible. There was also a realistic prospect of shaping future work so as to implement the missing components for points 3 and 4 above, or at least receiving guidance in how to address those deficiencies.

3.5 Experimental validation of parallelisation approach

With both the mutation tool and the means to accelerate its operation chosen, it seemed a good idea to validate the latter decision. Having implemented an early version of the wrapper script, hereafter known as MultiMutiny, it was possible to put its parallelisation algorithm to the test with a small experiment. I created a second script known as `mm-benchmark` (and available at https://github.com/timw6n/mm-benchmark) which would create multiple copies of the program code and associated specs for the Mutiny demo application described in Appendix 2. Each newly inflated program, comprising between 1 and 50 such copies, would then be tested three times with MultiMutiny to produce an average execution time. This is perhaps not enough data, or a sufficiently realistic methodology, for a truly rigorous experiment, but it was only really intended to confirm that the approach was feasible.

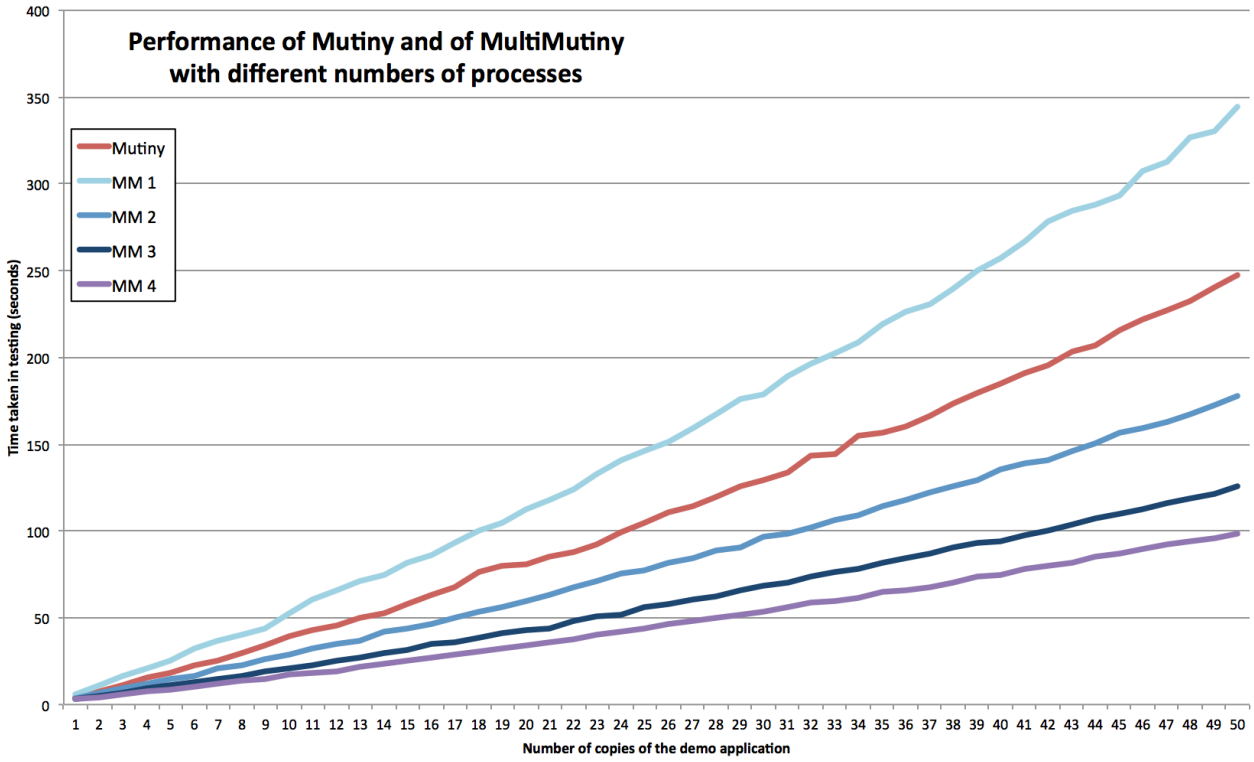

Figure 3: A graph showing the time required to mutation test different numbers of program copies. Times are given for Mutiny and for MultiMutiny using 1–4 processes. Results for 5–8 processes are not shown, but would closely follow the four-process line.

The key results arising from this experiment, as run on my quad-core desktop computer, were as follows, in roughly decreasing order of significance:

- Parallelisation via MultiMutiny does provide quite a significant speed increase, versus running Mutiny on its own. This is especially true as the size of the program-under-test increases: As the graph in Figure 3 shows, at 50 copies of the `Palindrome` class and related tests and using four processes, the testing process took just under 40% of the usual time.
- Even with quite trivial programs, parallelisation still provides a speed increase: For the unmodified example program (i.e. the point relating to one copy on the graph) the four-core parallelised implementation took, on average, only 74% of Mutiny's normal runtime. With two scoring processes in use, there was still a time-saving of around 10% shown in the same circumstances.

- There was a certain amount of overhead associated with wrapping Mutiny in this way: Running MultiMutiny in single-process mode caused the testing process to be around 40% slower. This is a legitimate use-case, given that MultiMutiny's CI and reporting features are likely to be useful (and hopefully considered worth the time penalty) even if either the program-under-test or the system hardware cannot support parallel operation.
- There was no appreciable speed increase beyond four simultaneous processes. This was as expected for a system with a quad-core CPU, as such a processor can only execute a maximum of four instructions at once.

I considered these findings sufficiently promising as to justify carrying on with the parallelisation approach.

3.6 Decision on how to set the mutation threshold

The final point to consider is how to set the mutation score threshold beyond which a given test is deemed to have passed. Using a simple, user-supplied threshold was the easiest possibility to implement and was the method eventually chosen. In the absence of any clear literature giving a clue as to an appropriate threshold level, I felt it wise not to make an editorial decision on this point, and to let MultiMutiny's eventual users make their own decision with their own codebases in mind.

An interesting alternative would have been to implement a "not decreasing" threshold, whereby each build for a project must match or exceed the previous build's score. The problem with implementing this came from the fact that, as described above, MultiMutiny must not rely on persisting any state on the machine running it. It would be necessary therefore for the cloud component to have a concept of a project, and of a sequence of builds within a project, something which, though by no means impossible, would mean that it would end up taking over some of the responsibilities of the CI systems triggering it. This would violate the engineering principle of doing "one thing well". Besides, it would be by no means impossible for an end-user to build such a system on top of MultiMutiny, if running it on machine with non-temporary storage available.

4 Design & Implementation

The main deliverable from this project is the MultiMutiny application, which is composed of two components; the main application, which runs on the computer performing the mutation testing and a separate cloud-based reporting component. This chapter will describe both of these parts, and, briefly, the development process that created them, before describing the scoring parallelisation algorithm in greater detail.

Both sides to the program are written in Python, with the client-side component using Python 3 and the cloud-side application hosted on Google App Engine and therefore using that platform's integrated Python 2.7 runtime. The relevant code (and a commit history showing how development progressed) is contained within the GitHub repositories in the table below, and so only one small code extract will be quoted in this chapter.

MultiMutiny	https://github.com/timw6n/multimutiny	468 lines of code
MultiMutiny Cloud Reports	https://github.com/timw6n/mm-cloud-reports	222 lines of code

The line counts above are intended to show the relative sizes of the two projects, and include Python files and HTML template files only. They were calculated using `git ls-files "*.py" "*.html" | xargs wc -l`.

4.1 Development process and software lifecycle

As mentioned at the beginning of the previous chapter, I followed a highly iterative approach to developing the software for this project: Requirements were decided upon, and functionality added, mostly on the fly.

Dogmatically and completely applying a set Agile workflow, such as Scrum or Kanban[41], would be somewhat excessive for a project such as this, with only myself as both developer and project manager combined. I did however attempt to apply as many Agile principles as were relevant and sensible, given the circumstances. It has already been mentioned that the development process was highly iterative, with a working, but basic, version of MultiMutiny produced quite early on in the year and then refined. Also, one could liken the time between each supervision to a week-long sprint, with supervisor meetings functioning as a quasi-retrospective and planning session. That said, it would be fair to say that my plans for each week never reached the level of fine detail that would be required for, for instance, tickets in a real-world team's issue tracker.

Somewhat ironically, given this project's focus on building quality automated test suites, no automated tests were written for MultiMutiny. The application was, however, tested quite extensively against various sample programs (chiefly the Mutiny demo application described in Appendix 2), and, as will be shown in the Evaluation, doesn't seem to suffer from any major functional bugs. The main factor behind the lack of automated tests was the availability of time, as is often the case with academic software, coupled to the fact that features are much more noteworthy in reports like this one. Secondary to that was the fact that an automated test suite for MultiMutiny alone would not prove particularly useful: Most of the bugs encountered during development were in the mutation tool, rather than in the wrapper around it. Obviously, the absence of an automated test suite precludes the application of mutation testing to MultiMutiny itself.

4.2 Application components and features

4.2.1 The client-side application

The basic structure of, and tasks performed by, the main, client-side part of the MultiMutiny application is given in the diagram in Figure 4 overleaf. This section will give much more details regarding that structure, and the justifications behind it, starting with the relationships between the parts represented by the differently-coloured shapes.

Recall that, in order for the results of this project to be as widely applicable as possible, MultiMutiny was intended to be coupled as loosely as possible to Mutiny, the mutation tool for which it was initially implemented. All of the code that communicates, via shell commands, with Mutiny is confined to a single, unsurprisingly named, `Mutiny` class shown in Figure 5, which responds to a set interface that should be applicable to other mutation tools. Certainly, there is no code in the main application that requires a specific tool, and the architecture is there to enable wrapper classes for other tools to be dropped-in and selected via MultiMutiny's `-t` command-line flag. One point at which, however, there is something of a closer, but implicit, coupling is in the reading of mutants and their metadata from the relevant `.mutants` directory. While neither this directory's

name, nor its contents, mention Mutiny by name, MultiMutiny does expect the relevant files to be in the format produced by Mutiny. I do not consider this to be a huge problem however, as a hypothetical wrapper class for another tool could always manipulate the filesystem itself if some light format-shifting were required.

A similar desire for extensibility led to the use of an almost identical plugin-type structure for what I've termed scorers and reporters, represented on the diagram by green pentagons and ClipArt-style icons respectively. There is only one scorer presented as part of this report; a `LocalScorer` class, each instance of which creates a temporary directory on the local filesystem, copies the project-under-test there, together with that scorer's subset of all the generated mutants, and then instructs the mutation tool to score those mutants. The assignment of mutants to scorers is controlled by the parallelisation algorithm defined below, with the parallel execution itself being by way of the mapping functionality provided by the `multiprocessing` package in Python's standard library. The concept of scorers was split-out from the main application code to provide a clear path towards the creation of a distributed scoring system, an idea explored further in the Conclusion chapter.

Reporters, for their part, receive the parsed mutation results, with mutant metadata already added, and are responsible for communicating those results, in one form or another, to the user. Three reporters are provided as part of this project; one which simply outputs to the command-line, one which generates a standalone HTML file, and one which communicates with MultiMutiny's cloud component and is ideally-suited to being used with cloud-based CI systems.

Screenshots of reports produced by the CLI and HTML reporters are given as Figures 6 and 7 respectively. The majority of the details given in these reports should be self-explanatory, but one term that needs defining is the "pessimistic mutation score". The word pessimistic has been prepended to emphasise that neither Mutiny nor MultiMutiny make any effort to identify equivalent mutants: The real mutation score, if one accepts that it should not take equivalent mutants into account, could be higher than the figure given. The HTML reports are similar in all but one aspect to those presented by the cloud component, and so will be covered, along with the cloud reporter, in more detail in the next subsection.

Finally, it is important to give a brief description of MultiMutiny's benchmark mode, which was utilised when producing the timings given in the Evaluation chapter. As we have seen, MultiMutiny has more features than, and therefore needs to do more work than, Mutiny on its own. The addition of the benchmark mode is an attempt to make timing measurements taken against MultiMutiny somewhat comparable with those taken against Mutiny, and thereby allow the efficacy of the parallelisation procedure to be judged. Enabling benchmark mode disables the potentially time-consuming loading of mutant metadata from disk, as well as the project compatibility check normally performed at the start of each run.

4.2.2 The cloud reporting system

As mentioned at the head of this chapter, MultiMutiny's cloud component was built as a separate codebase to the main application and runs on Google's App Engine. App Engine was chosen because it abstracts away many sysadmin-type concerns, making the cloud reporting system highly-available, fast and scalable without me having to, for example, set-up and maintain any VPS instances. Using Python, the same language as for the client-side application, albeit an older version, meant that some code and templates, particularly those involved in actually rendering reports, could be re-used from the standalone HTML reporter.

Figure 8 is a screenshot of a report displayed by the cloud system. The report is, as one might expect given the above comment regarding shared code, pretty much identical to an equivalent HTML report. This report was generated in CI mode, hence the large green "CI BUILD PASSED" text. Each mutant's name is a hyperlink that, when clicked, opens up a modal window similar to that shown in Figure 9. As one can see from the figure, that box gives a standard-format diff between the relevant original and mutant files. Originally, the full contents of each file were displayed side-by-side in the dialogue, which was fine for the very small files in toy applications, but proved somewhat impractical for larger, actually useful, projects. The links at the bottom of the dialogue point to full versions of both of those files. These links are not present for the standalone HTML reports as, when using that reporter, it was considered safe to assume that the user has access to the raw files within the `.mutants` directory.

Reports are uploaded, by the cloud reporter, to the cloud system as the JSON-format payload of an HTTP PUT request. An example set of results, as translated into JSON, is given in Appendix 3. Once received by the server, each uploaded report is validated against the JSON schema given in Appendix 4. A URL pointing to that report is then returned to the client, which outputs it to the command-line. Using a schema is quite a heavyweight solution, in comparison to simply assuming the object received has the correct structure and relying on exceptions being thrown if it doesn't, but does have some significant advantages: It enables report-handling

code protected by the schema check to be cavalier with regard to accessing keys and the like, in the knowledge that any report that code will touch must be of the expected format. Going forward, using a schema-based system could make it easier to maintain backwards compatibility with older clients, as, when new features are added, a new schema version could be created, with requests matched against the appropriate version. A well-defined schema also allows the possibility of additional clients being able to communicate with the cloud system.

4.3 MultiMutiny's parallelisation algorithm

MultiMutiny's parallelisation algorithm is extremely simple, and reproduced in full below. It is the method responsible for allocating mutants to scoring instances.

```
def divide(sequence, numsegments):
    return [sequence[i::numsegments] for i in range(numsegments)]
```

As the above code is quite concise, its function is easiest to explain using an example: Let's say that we have a list of five mutants, represented by variable names `a` to `e`, and wish to divide them across three parallel scoring processes, numbered `1`, `2` and `3`. This would lead to a method call of `divide([a, b, c, d, e], 3)`, which gives the result `[[a, d], [b, e], [c]]`, where each of the nested lists represents the mutants assigned to a scorer. The allocation method is even clearer when expressed in the tabular form that will be re-used in the Evaluation chapter:

1	2	3
a	b	c
d	e	

Clearly, this algorithm make no attempt to optimise the allocation with reference to any of the mutant's properties. Its performance was, however, deemed to be sufficient given the results of the experiment detailed in the Analysis chapter. Discussion of alternative, and potentially better, allocation methods will therefore be deferred until the Evaluation and Conclusion chapters.

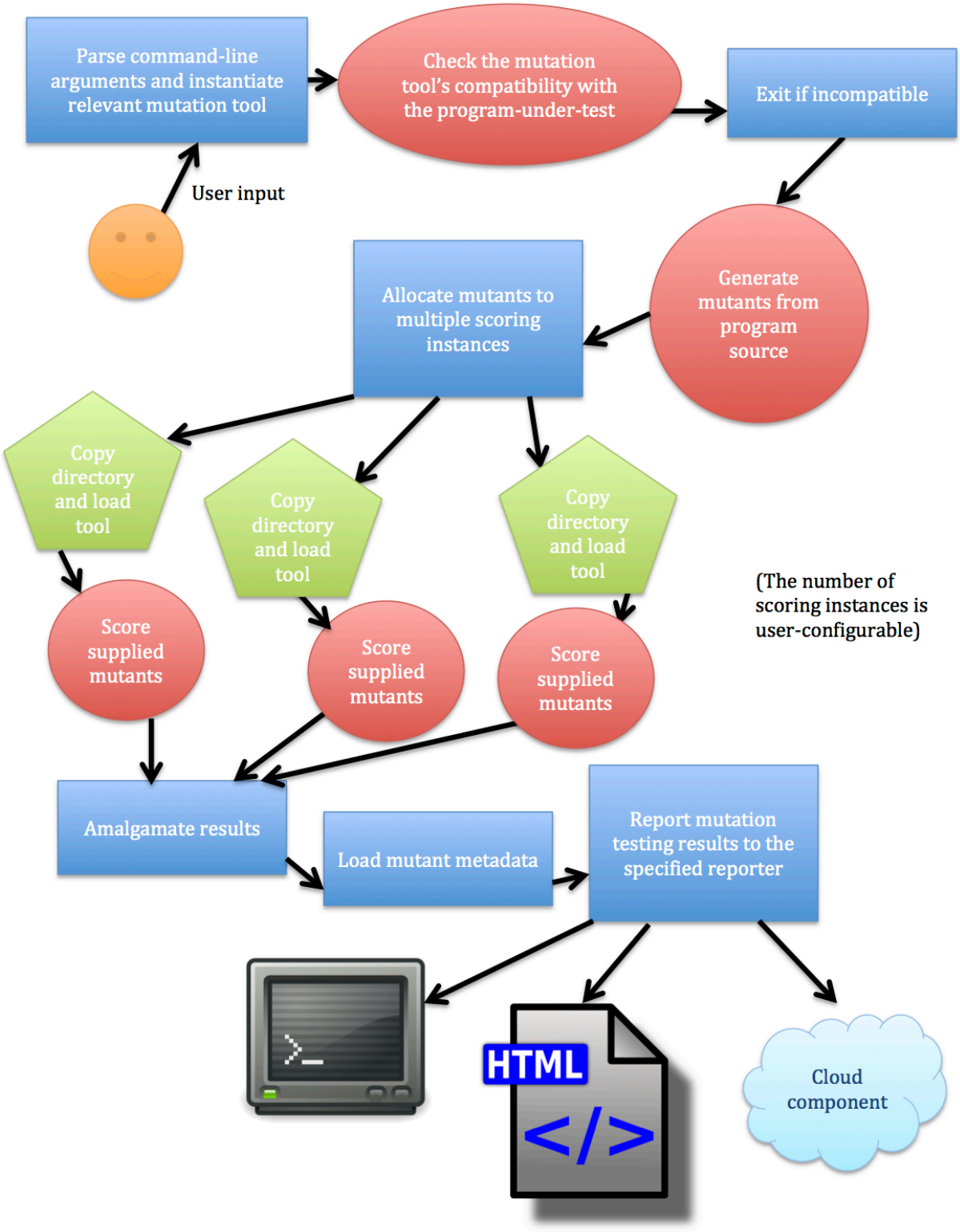

Figure 4: The basic structure of the MultiMutiny program. The blue rectangles represent steps carried out by the MultiMutiny core code, the green pentagons steps carried out by the scoring instances and the red circles those performed by the mutation tool. The three icons at the bottom represent the three reporting components.

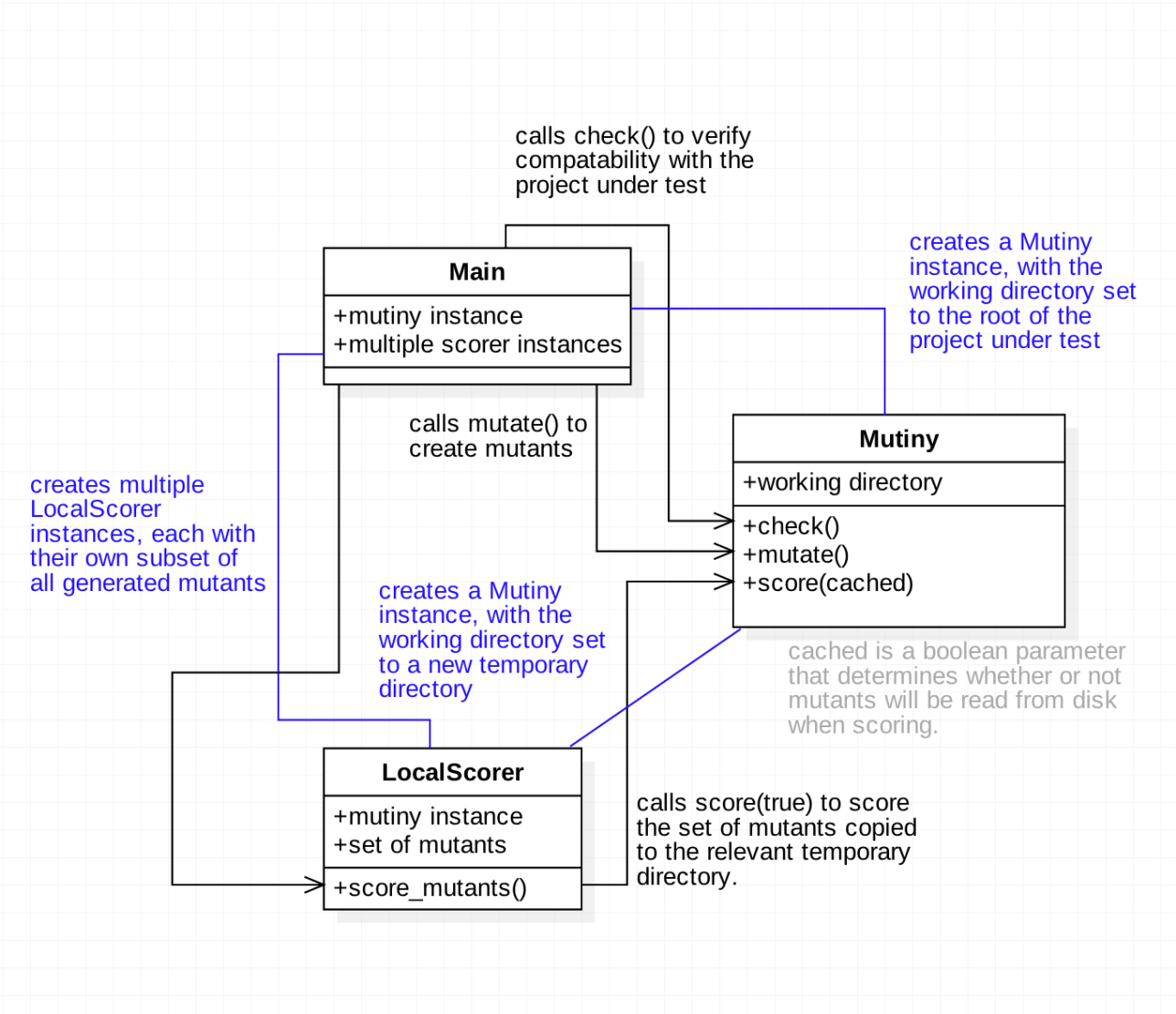

Figure 5: The Mutiny class and those classes that instantiate it. Attributes and methods for other classes are given only where they have some connection to the Mutiny class. Blue lines show instance creation, black arrows method calls to a created instance.

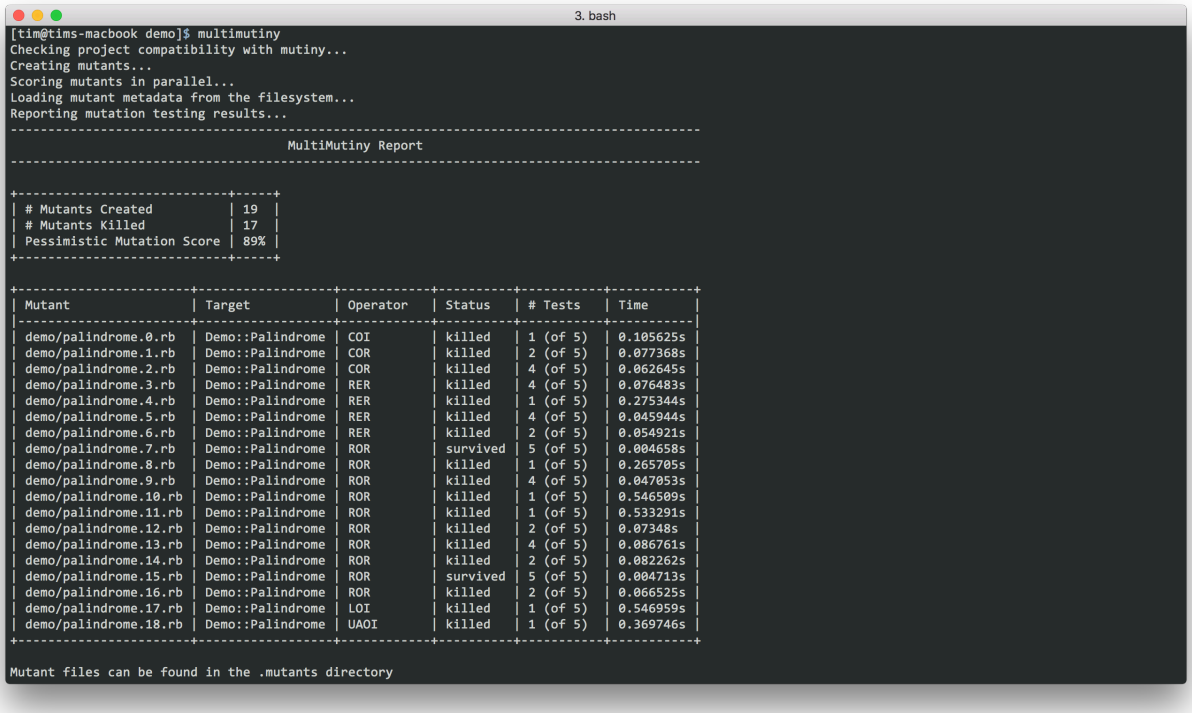

Figure 6: MultiMutiny CLI output for the Mutiny demo application.

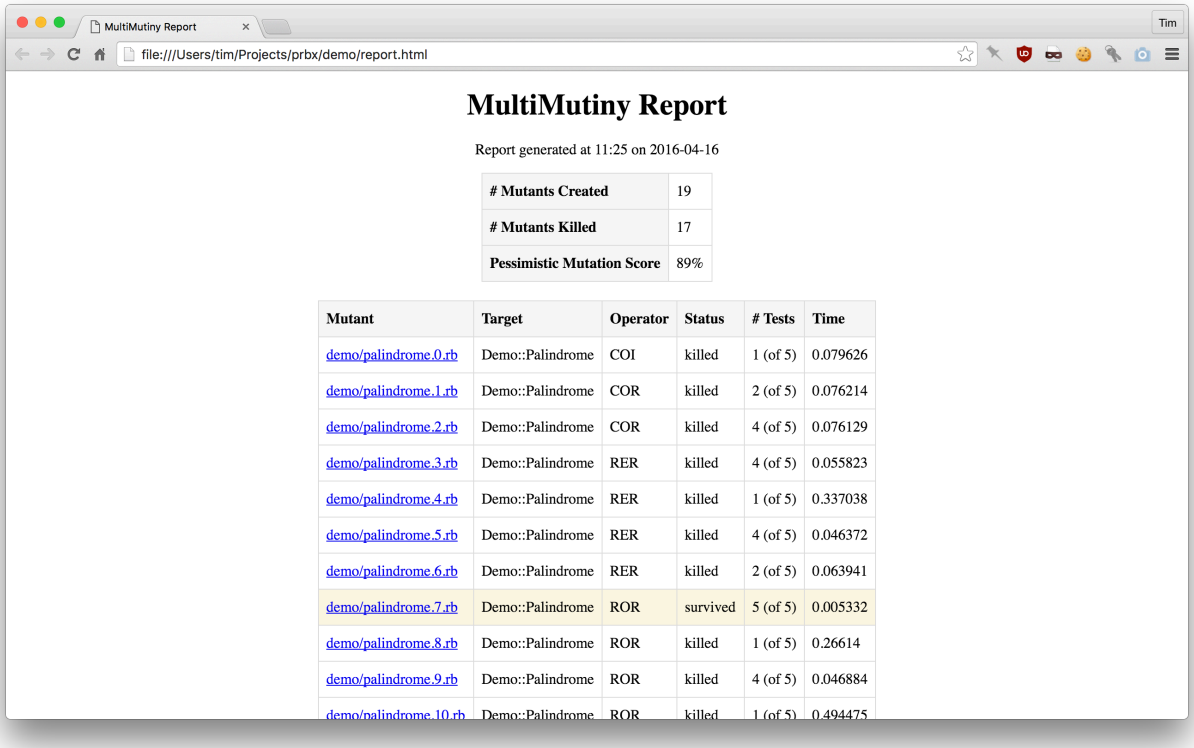

Figure 7: A local HTML version of the above report.

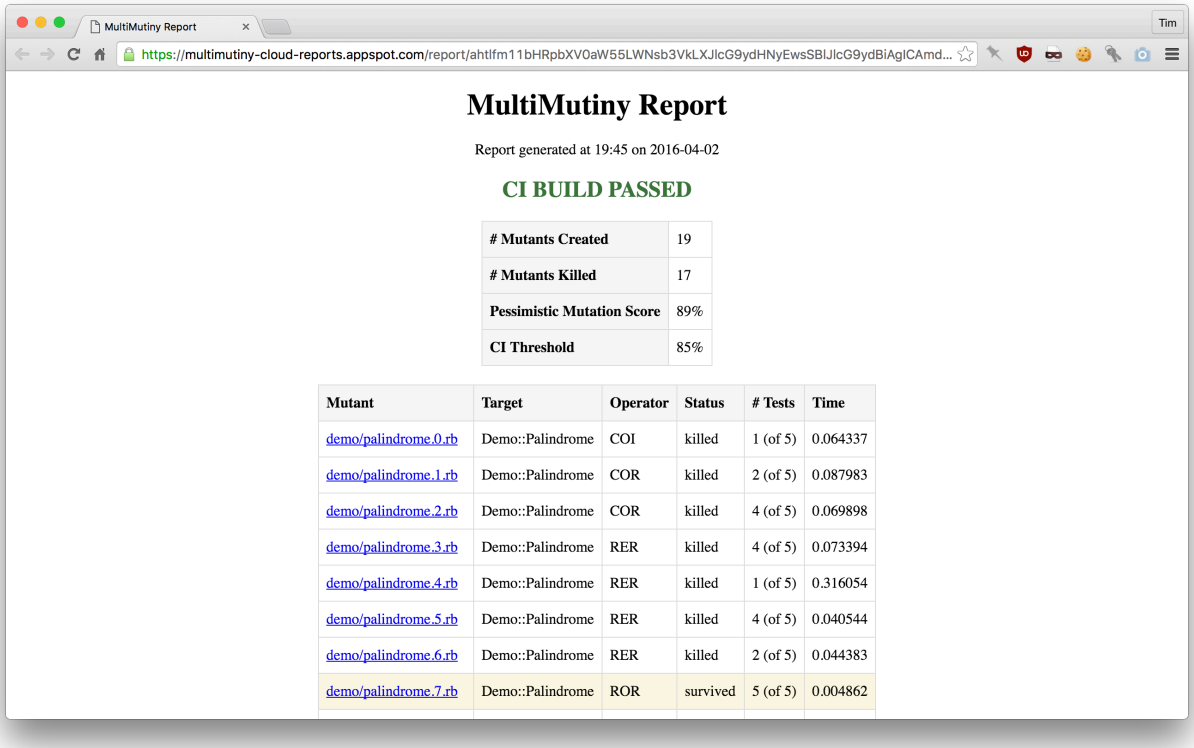

Figure 8: A MultiMutiny cloud report in CI mode. This build has passed as the pessimistic mutation score of 89% was above the 85% threshold set.

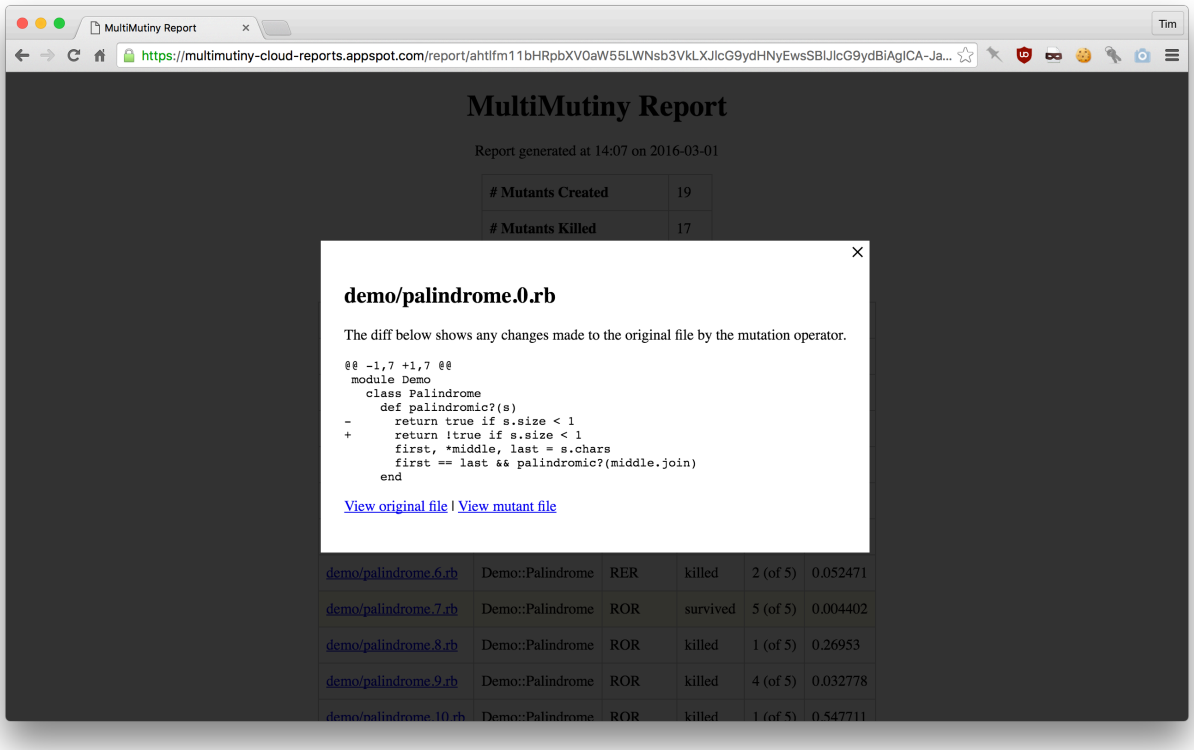

Figure 9: The same cloud report as the above figure, showing more details for the first mutation.

5 Evaluation

The overarching goal of this project is to enable the benefits of mutation testing to be realised for real codebases, through the medium of CI, so it seems appropriate to evaluate it with respect to some existing open-source projects. The three such projects used throughout this chapter are detailed in the table below.

Project	GitHub repository	Description (from GitHub)
event_bus	kevinrutherford/event_bus	A simple pubsub event bus.
full_name	agilidee/full_name	Adds `full_name` method for classes that provide a `first_name` and a `last_name`!
lumberjack	bdurand/lumberjack	A simple, powerful, and very fast logging utility that can be a drop in replacement for `Logger` or `ActiveSupport::BufferedLogger`.

These projects were chosen from a list of projects amenable to mutation testing provided in Mutant's documentation[42], and from those projects' dependencies. Many codebases were eliminated from consideration because they were unsupported by Mutiny, which was at version 0.3.0 at the time. Mutiny's lack of compatibility does limit somewhat the applicability of this project but it is important to remember that MultiMutiny is, to a certain extent, independent of the underlying tool and that, as bugs in Mutiny are found and fixed, the pool of potential MultiMutiny targets will expand automatically.

5.1 Functionality

This section will be quite brief because, in short, MultiMutiny works as expected. Applying it to the three projects under consideration yielded the results summarised in the table below. Complete copies of the command-line report arising from each project are given in Appendix 5, with a build without CI features, a passing CI build and a failing CI build each being demonstrated. Links to uploaded versions of HTML reports and to live reports on the cloud system are given in Appendix 6.

Project	Mutants generated	Mutants killed	Pessimistic mutation score
event_bus	57	51	89%
full_name	51	51	100%
lumberjack	565	480	85%

It is also quite clear that the software provides sufficient affordances as to allow it to operate as desired when called from CI programs. What should be highlighted in particular, given that it isn't evidenced in the reports in the two appendices, is that the correct exit codes are set to signal whether a given CI build has passed or failed.

As the results from the lumberjack project in particular demonstrate, MultiMutiny can cope with reasonable-sized projects and with a relatively large number of mutants. In fact, I can think of no hard limit as to the number of mutants that can be accommodated by the CLI and local HTML reporters.

For the cloud reporter, there are two limits set by the underlying Google App Engine platform that may prove to be problematic as the mutant count increases: The first of these is a 32MB limit imposed in each incoming HTTP request[43], including the PUT requests used to transmit reports to the server. This shouldn't prove a huge problem given that it is quite a high limit and that compression could quite easily be added to requests: Given that the JSON reports contain a large amount of repeated data, with each mutant's entry containing a copy of the original and mutated files for instance, a significant reduction in file size should be easily achievable. The JSON file generated by testing the lumberjack project is 5.7MB (so still comfortably below the limit) uncompressed, and applying zlib compression reduces it to only 67KB.

A 1MB limit on datastore entries[44] may prove to be more of a problem, as each report is presently stored as a single entry. The lumberjack report would exceed this limit, were compression not applied. It would not be massively complicated however, were the need to arise, to split reports up into multiple entities, perhaps one per mutant scored, and thereby remove this worry entirely.

One piece of functionality that MultiMutiny's cloud component is lacking, when compared to commercial code

coverage tools such as Coveralls[36] and Codecov[45], is the provision of aggregate statistics for a project. Figures 10 and 11 give different visualisations, from these two products, of the progression in scores for example repositories. It would not be a huge technical undertaking to add something similar to MultiMutiny though: As can be inferred from Appendix 3, all the necessary information is already stored in a machine-queryable format, and the existing API keys could be repurposed to serve as a target project's unique identifier. The more challenging task would be to determine which statistics are actually valuable to developers, and the format in which to present them.

5.2 Speed

One of the main aims of this project was to make a contribution to speeding-up the mutation testing process, so as to make it more practical for CI use. For the reasons given in the Analysis chapter, I chose to do this by parallelising the mutant scoring process. The table below gives the average times taken in testing each of the three projects mentioned above, as performed by Mutiny and by MultiMutiny (in benchmark mode) using four parallel processes.

Project	Mutiny	MultiMutiny	MultiMutiny as a % of Mutiny
event_bus	17.32s	7.20s	41.59%
full_name	15.30s	6.55s	42.79%
lumberjack	142.68s	51.78s	36.29%

All timings in the above table were recorded on the same quad-core machine as was used to create the benchmarks in the Analysis chapter. MultiMutiny was run in benchmark mode and four scoring processes were specified. Times given are in seconds, are rounded to two decimal places and are mean averages from five runs against each project.

As can be seen from these results, the speed increase afforded by MultiMutiny's parallelisation system is both significant and broadly in-line with (or slightly better than) that expected given the results recorded in the Analysis chapter. To put these results into context, recalling the aforementioned "ten minute rule" for useful CI builds, this is the equivalent of bringing projects that would normally take up to around 27½ minutes to mutation test within that limit.

That said, and as noted earlier, the algorithm used is rather naive and thus far from optimal. The next part of this section will attempt to quantify that sub-optimality somewhat, by means of the results (and in particular the timings for each mutant) returned for the run against the full_name project shown in Appendix 5.

The table overleaf shows the actual allocation of mutants to scoring processes (and thus processor cores) produced by MultiMutiny in that run[1], together with the time taken in scoring each mutant. That time includes only that spent on the body of the tests, not that setting up or loading them, which is why the totals (in bold) end up being much less than the actual runtime given previously. Those overheads should at least be more or less constant for each mutant however, so shouldn't affect the validity of the conclusions drawn.

[1]The reason why the allocation may seem slightly odd in the light of the explanation given in the previous chapter is that, at the point where allocation takes place, the mutants are in strict alphabetical order, i.e. 1 is followed by 10 rather than 2.

1	2	3	4
0 (0.04193s)	1 (0.042936s)	4 (0.040465s)	3 (0.039877s)
2 (0.044003s)	5 (0.041161s)	9 (0.035455s)	6 (0.038068s)
7 (0.036082s)	8 (0.03985s)	10 (0.038636s)	11 (0.038325s)
12 (0.039776s)	13 (0.046286s)	14 (0.039437s)	15 (0.027894s)
16 (0.040467s)	17 (0.032095s)	18 (0.027813s)	19 (0.027666s)
23 (0.028366s)	20 (0.042332s)	21 (0.027737s)	22 (0.028771s)
27 (0.028093s)	24 (0.029405s)	25 (0.029342s)	26 (0.026229s)
30 (0.026223s)	28 (0.030884s)	29 (0.027865s)	33 (0.026222s)
34 (0.029065s)	31 (0.027676s)	32 (0.026059s)	37 (0.028659s)
38 (0.026948s)	35 (0.032234s)	36 (0.027367s)	40 (0.027033s)
41 (0.028182s)	39 (0.028032s)	43 (0.028302s)	44 (0.027506s)
45 (0.027445s)	42 (0.028905s)	47 (0.026493s)	48 (0.02702s)
49 (0.029148s)	46 (0.026962s)	50 (0.02864s)	
0.425728s	**0.448758s**	**0.403611s**	**0.36327s**

This gives a total execution time of 0.448758 seconds. It is important to note however that, were the mutants generated by Mutiny in a different order, the worst possible allocation that could be achieved (namely the 13 most time-consuming mutants assigned to the same processor) would lead to a time of 0.537156 seconds.

With a list of jobs (each mutant requiring scoring) and their execution times, finding an optimal allocation with which to compare the above allocations is a classic example of the multiprocessor scheduling problem. A complete description of that problem would be beyond the scope of this section but it suffices to state that it is a NP-complete problem and that the longest processing time-first (or LPT) algorithm provides a reasonable approximation[46]. LPT scheduling requires sorting the jobs in non-increasing execution time order and assigning each job, in turn, to the processor with the earliest expected finish time[46]. Applied to the set of mutants above, it gives the allocation below.

1	2	3	4
4 (0.040465s)	2 (0.044003s)	1 (0.042936s)	0 (0.04193s)
7 (0.036082s)	6 (0.038068s)	3 (0.039877s)	8 (0.03985s)
9 (0.035455s)	12 (0.039776s)	5 (0.041161s)	11 (0.038325s)
13 (0.046286s)	16 (0.040467s)	10 (0.038636s)	15 (0.027894s)
14 (0.039437s)	19 (0.027666s)	21 (0.027737s)	17 (0.032095s)
18 (0.027813s)	22 (0.028771s)	23 (0.028366s)	20 (0.042332s)
26 (0.026229s)	29 (0.027865s)	24 (0.029405s)	25 (0.029342s)
27 (0.028093s)	33 (0.026222s)	28 (0.030884s)	30 (0.026223s)
34 (0.029065s)	35 (0.032234s)	32 (0.026059s)	31 (0.027676s)
44 (0.027506s)	40 (0.027033s)	38 (0.026948s)	36 (0.027367s)
48 (0.02702s)	43 (0.028302s)	39 (0.028032s)	37 (0.028659s)
50 (0.02864s)	46 (0.026962s)	42 (0.028905s)	41 (0.028182s)
	49 (0.029148s)	45 (0.027445s)	47 (0.026493s)
0.392091s	**0.416517s**	**0.416391s**	**0.416368s**

This gives a total execution time of 0.416517 seconds, a modest saving in absolute terms when compared to the actual and worst-case allocations but a reasonably significant one when expressed in percentage form: A saving of just over 7% is achieved against the actual allocation, with a more dramatic saving of just over 23% against the worst-case scenario. Both of these percentages represent much smaller increases in performance than the one achieved by moving from Mutiny to MultiMutiny.

Of course, it would be unreasonable to expect MultiMutiny to be able to achieve such an allocation without advance knowledge of the mutant scoring times. However, as will be expanded upon in the Conclusion chapter, it may well be possible to utilise saved state from previous scoring runs to approximate the application of the approximation algorithm given above.

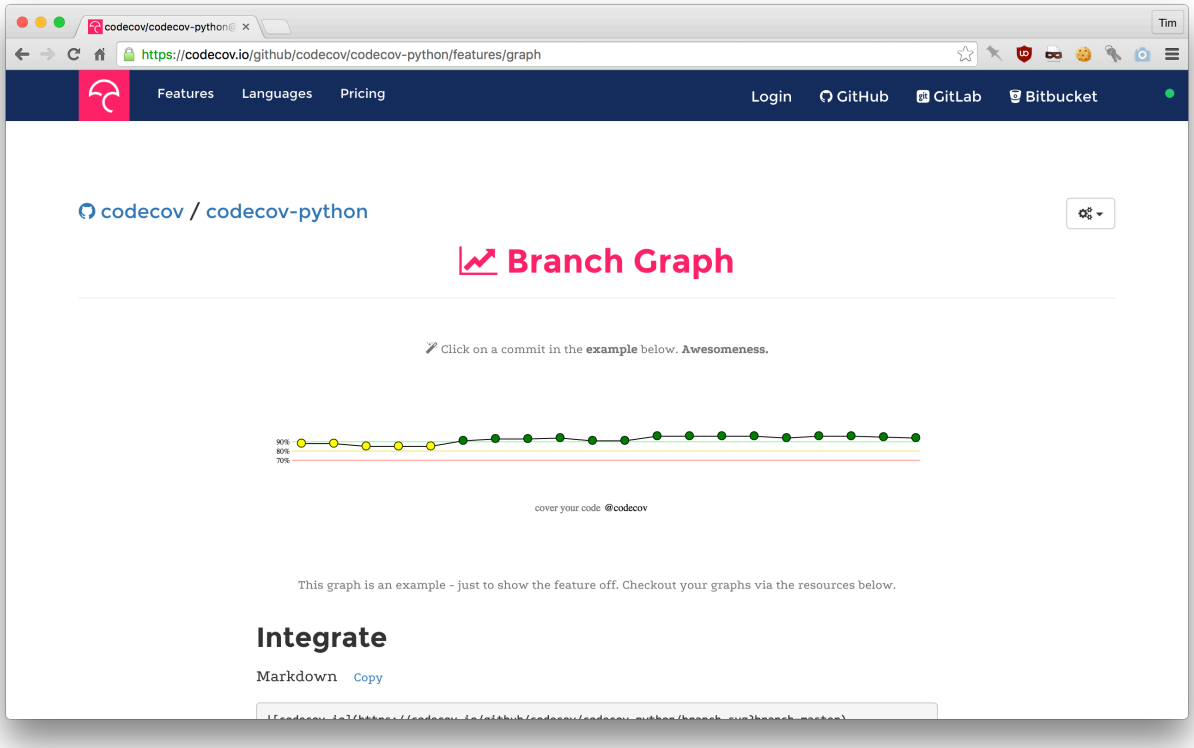

Figure 10: An example coverage progression graph, as produced by Codecov.

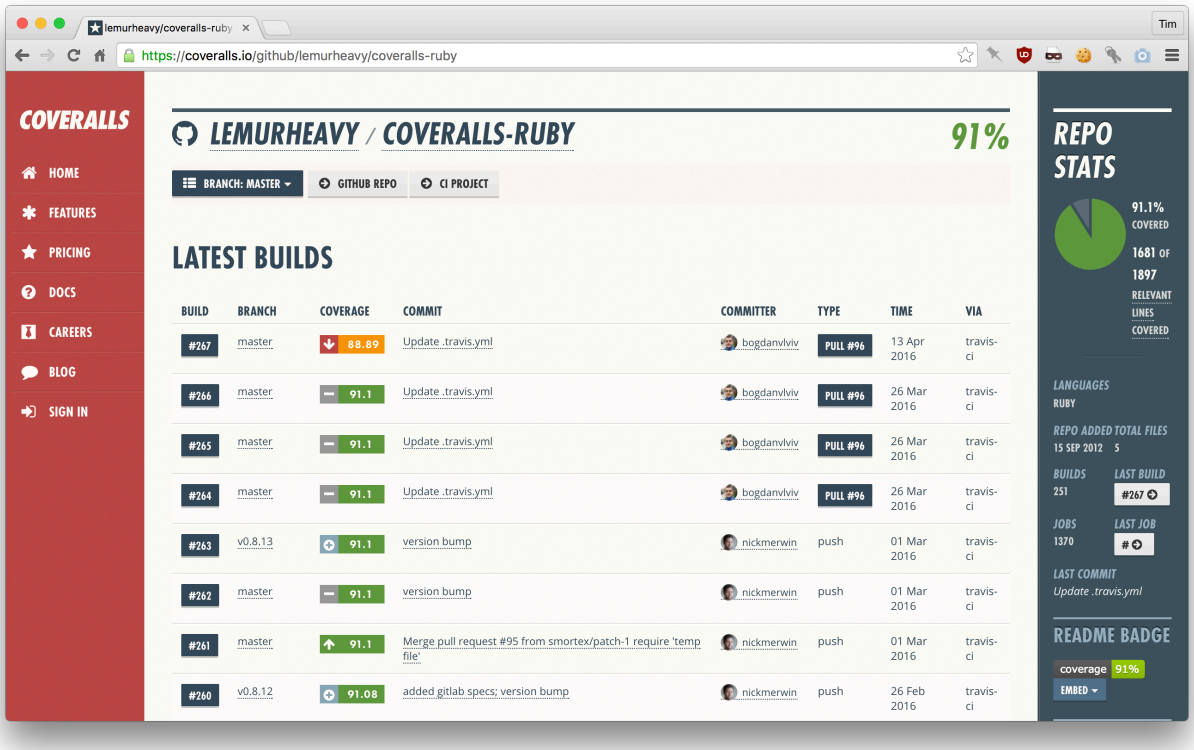

Figure 11: Coverage changes for another example repository, as shown by Coveralls.

6 Conclusion

The main contribution of this project has been to provide software capable of running mutation testing against real projects, with acceptable performance, and generating reports suited to a CI environment.

To backtrack a little, the Literature Review, after introducing some basic concepts, highlighted the requirement for processes run as part of a CI build to be reasonably speedy, and gave details of why mutation testing has struggled to meet that need, as well as some potential approaches to remedying that deficiency. There was also a discussion as to how the percentage scores produced by mutation testing could be translated into the binary result required from CI builds.

The Analysis chapter outlined the decisions made in implementing this project's software, in particular the choice to wrap around an existing mutation tool, and to use parallelisation to increase that tool's performance. The Design & Implementation chapter gave more details as to how those decisions were implemented, with a particular focus on the manner in which the application was designed with extensibility in mind. Finally, the Evaluation chapter proved that MultiMutiny can work with real projects, and that, in doing so, it achieves a time-saving of around 60% of the time that, without the benefit of the wrapper this project provides, Mutiny would require.

6.1 Further work

There are several facets of this project that would benefit from further work, particularly in relation to improving the process of scoring parallelisation, to speeding up the system more generally, and to using a better method for setting the mutation threshold in CI mode.

To tackle the first of these, it has been shown in the previous chapter that MultiMutiny's current parallelisation algorithm, while adequate for achieving a reasonable speedup, is not entirely optimal. It may be better to use a work-queue model, whereby, instead of allocating tasks to processes upfront, a queue of mutants is kept, from which idle scoring processes would pick. While this would require more code and greater inter-process communication than there is at present, it does not seem like it would be an enormous change and could help avoid some of the most pathological possibilities inherent in the current system. Splitting the running of a program's test suite between cores could save even more time, but would be much more challenging, if at all possible. Another interesting possibility would be to retain ahead-of-time allocation, but to use data (particularly the time taken in scoring each mutant) from previous runs to inform the allocation. In the case where the program-under-test is more-or-less unchanged, for builds where only variable names or comments have changed for instance, this would almost certainly result in an optimal allocation.

Another possibility with regard to parallelisation would be to split the work across multiple machines. This could be implemented over SSH, with the hypothetical `RemoteScorer` ideally responding to the same interface as the current `LocalScorer`. Distributing the scoring workload in this way would mean that, assuming an infinite availability of machines (which is effectively the case for platforms like Amazon EC2, if one's budget is sufficiently robust) and discounting network overheads, mutation scoring could be performed within the maximum time taken for a single execution of the relevant test suite. This approach would also mean that programs that cannot co-exist with themselves on a single machine (e.g. applications that always bind to a specific port) could still be scored in parallel.

Another possibility for speeding-up the system, at least when running in CI mode, is to halt execution once the outcome of the build is known, i.e. to fail fast. For instance, if the mutation threshold were 60%, scoring could cease when either more than 40% of the tests have failed or 60% have passed. While this approach could result in a marginal speed increase in certain circumstances, it is not as promising as other approaches to this problem: This method has no impact whatsoever on the worst-case runtime, and, assuming that the threshold is set such that most builds are expected to pass, the average speed-up will be marginal. Moreover, users may prefer to be presented with a complete set of mutation results, and an accurate score, rather than a range of possible scores.

The final area for further work identified was in using the mutation score output from the previous test run as the mutation threshold for the following run, thereby mandating a non-decreasing mutation score. This feature was omitted for the reasons given in the Analysis chapter but, if it were decided to increase MultiMutiny's scope to make it the mutation testing equivalent of code coverage products like Coveralls or Codecov, then adding this feature would certainly be a good idea.

Appendix 1: Table of ASCII Characters

The following table lists each of the 128 standard ASCII characters, together with their respective integer code.

0	(nul)	32	(sp)	64	@	96	‘
1	(soh)	33	!	65	A	97	a
2	(stx)	34	"	66	B	98	b
3	(etx)	35	#	67	C	99	c
4	(eot)	36	$	68	D	100	d
5	(enq)	37	%	69	E	101	e
6	(ack)	38	&	70	F	102	f
7	(bel)	39	'	71	G	103	g
8	(bs)	40	(	72	H	104	h
9	(ht)	41	)	73	I	105	i
0	(nl)	42	*	74	J	106	j
11	(vt)	43	+	75	K	107	k
12	(np)	44	,	76	L	108	l
13	(cr)	45	-	77	M	109	m
14	(so)	46	.	78	N	110	n
15	(si)	47	/	79	O	111	o
16	(dle)	48	0	80	P	112	p
17	(dc1)	49	1	81	Q	113	q
18	(dc2)	50	2	82	R	114	r
19	(dc3)	51	3	83	S	115	s
20	(dc4)	52	4	84	T	116	t
21	(nak)	53	5	85	U	117	u
22	(syn)	54	6	86	V	118	v
23	(etb)	55	7	87	W	119	w
24	(can)	56	8	88	X	120	x
25	(em)	57	9	89	Y	121	y
26	(sub)	58	:	90	Z	122	z
27	(esc)	59	;	91	[	123	{
28	(fs)	60	<	92	\	124	|
29	(gs)	61	=	93	]	125	}
30	(rs)	62	>	94	^	126	~
31	(us)	63	?	95	_	127	(del)

Source: [47]

Appendix 2: Mutiny demo application

The following two code listings show the main body and spec file for the Mutiny demo application. The purpose of the one method provided is to return `true` if a supplied string is a palindrome, and `false` otherwise. A palindrome is a word that is unchanged when the order of its letters are reversed, for example "detartrate".

This demo application was created by Dr Louis Rose to provide an example codebase that would be supported by Mutiny. The full project repository (including the two files quoted below) is available at https://github.com/timw6n/mutiny-demo, which is itself a fork of https://github.com/mutiny/demo.

palindrome.rb

```
module Demo
  class Palindrome
    def palindromic?(s)
      return true if s.size < 1
      first, *middle, last = s.chars
      first == last && palindromic?(middle.join)
    end
  end
end
```

palindrome_spec.rb

```
module Demo
  describe Palindrome do
    it "should accept the empty string" do
      expect(subject.palindromic?("")).to be_truthy
    end

    it "should accept a short palindrome" do
      expect(subject.palindromic?("aa")).to be_truthy
    end

    it "should accept a longer palindrome" do
      expect(subject.palindromic?("baab")).to be_truthy
    end

    it "should reject a short non-palindrome" do
      expect(subject.palindromic?("ab")).to be_falsey
    end

    it "should reject a longer non-palindrome" do
      expect(subject.palindromic?("babb")).to be_falsey
    end
  end
end
```

Appendix 3: Example JSON report for the Mutiny demo application

A rather heavily abbreviated version of the JSON report sent to the cloud component after testing the Mutiny demo application (as described in Appendix 2) is given below. Only a representative sample of the 19 mutants generated are given, and, within the entry for each mutant, the full contents of the original and mutant files have been elided. The schema governing this document follows in Appendix 4.


```
{
    "timestamp": 1461060110,
    "results": [
        {
            "total_tests": 5,
            "time": 0.039179,
            "target": "Demo::Palindrome",
            "status": "killed",
            "name": "demo/palindrome.0.rb",
            "original": "[removed for brevity]",
            "mutant": "[removed for brevity]",
            "operator": "COI",
            "tests": 1
        },
        {
            "total_tests": 5,
            "time": 0.038299,
            "target": "Demo::Palindrome",
            "status": "killed",
            "name": "demo/palindrome.1.rb",
            "original": "[removed for brevity]",
            "mutant": "[removed for brevity]",
            "operator": "COR",
            "tests": 2
        },
        {
            "total_tests": 5,
            "time": 0.037137,
            "target": "Demo::Palindrome",
            "status": "killed",
            "name": "demo/palindrome.2.rb",
            "original": "[removed for brevity]",
            "mutant": "[removed for brevity]",
            "operator": "COR",
            "tests": 4
        },
        {
            "total_tests": 5,
            "time": 0.002894,
            "target": "Demo::Palindrome",
            "status": "survived",
            "name": "demo/palindrome.7.rb",
            "original": "[removed for brevity]",
            "mutant": "[removed for brevity]",
            "operator": "ROR",
            "tests": 5
        }
    ],
    "killed_mutants": 17,
    "total_mutants": 19,
    "percentage_score": 89,
    "ci": false
}
```

Appendix 4: Schema for JSON reports

The schema below follows the specification given at http://json-schema.org/documentation.html.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Schema for MultiMutiny reports",
  "type": "object",
  "properties": {
    "killed_mutants": {
      "type": "integer"
    },
    "total_mutants": {
      "type": "integer"
    },
    "percentage_score": {
      "type": "integer"
    },
    "timestamp": {
      "type": "integer"
    },
    "ci": {
        "type": ["boolean", "object"],
        "properties": {
            "passed": {
                "type": "boolean"
            },
            "threshold": {
                "type": "integer"
            }
        },
        "additionalProperties": false
    },
    "results": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "operator": {
            "type": "string"
          },
          "name": {
            "type": "string"
          },
          "status": {
            "type": "string",
            "pattern": "^(killed|survived|stillborn)$"
          },
          "tests": {
            "type": ["integer", "null"]
          },
          "total_tests": {
            "type": ["integer", "null"]
          },
          "time": {
            "type": ["number", "null"]
          },
          "target": {
            "type": "string"
          },
          "original": {
            "type": "string"
          },
          "mutant": {
            "type": "string"
          }
        },
        "additionalProperties": false
      }
    }
  },
  "additionalProperties": false,
  "required": ["timestamp", "killed_mutants", "total_mutants", "results", "percentage_score", "ci"]
}
```

Appendix 5: MultiMutiny reports for real codebases

This appendix gives complete reports for the real projects referred to in the Evaluation section, as produced by MultiMutiny's CLI reporter.

The first report in this section was produced in "normal" mode, i.e. without activating CI mode and setting a threshold, while the latter two reports show a passing and failing CI-mode build respectively.

The first line of each report (starting with `$`) gives the command run.

Links to uploaded copies of the equivalent HTML reports, and to reports live on the cloud reporting system follow in Appendix 6.

event_bus

```
$ multimutiny

-------------------------------------------------------------------------------------------------------------------
                                          MultiMutiny Report
-------------------------------------------------------------------------------------------------------------------

+----------------------------+-----+
| # Mutants Created          | 57  |
| # Mutants Killed           | 51  |
| Pessimistic Mutation Score | 89% |
+----------------------------+-----+

+------------------------------+-------------------------+------------+-----------+------------+-----------+
| Mutant                       | Target                  | Operator   | Status    | # Tests    | Time      |
|------------------------------+-------------------------+------------+-----------+------------+-----------|
| event_bus/registrations.0.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.017789s |
| event_bus/registrations.1.rb | EventBus::Registrations | COI        | killed    | 1 (of 31)  | 0.026732s |
| event_bus/registrations.2.rb | EventBus::Registrations | COI        | killed    | 10 (of 31) | 0.041409s |
| event_bus/registrations.3.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.016192s |
| event_bus/registrations.4.rb | EventBus::Registrations | COI        | killed    | 2 (of 31)  | 0.027454s |
| event_bus/registrations.5.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.017427s |
| event_bus/registrations.6.rb | EventBus::Registrations | COI        | killed    | 9 (of 31)  | 0.066081s |
| event_bus/registrations.7.rb | EventBus::Registrations | COI        | killed    | 10 (of 31) | 0.068974s |
| event_bus/registrations.8.rb | EventBus::Registrations | COI        | killed    | 8 (of 31)  | 0.036505s |
| event_bus/registrations.9.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.016032s |
| event_bus/registrations.10.rb | EventBus::Registrations | COI        | killed    | 1 (of 31)  | 0.038801s |
| event_bus/registrations.11.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.016618s |
| event_bus/registrations.12.rb | EventBus::Registrations | COI        | killed    | 1 (of 31)  | 0.047177s |
| event_bus/registrations.13.rb | EventBus::Registrations | COI        | killed    | 10 (of 31) | 0.043221s |
| event_bus/registrations.14.rb | EventBus::Registrations | COI        | survived  | 31 (of 31) | 0.015683s |
| event_bus/registrations.15.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.027413s |
| event_bus/registrations.16.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.035772s |
| event_bus/registrations.17.rb | EventBus::Registrations | LOI        | killed    | 10 (of 31) | 0.031001s |
| event_bus/registrations.18.rb | EventBus::Registrations | LOI        | stillborn | n/a        | n/a       |
| event_bus/registrations.19.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.03648s  |
| event_bus/registrations.20.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.027083s |
| event_bus/registrations.21.rb | EventBus::Registrations | LOI        | killed    | 9 (of 31)  | 0.030682s |
| event_bus/registrations.22.rb | EventBus::Registrations | LOI        | killed    | 10 (of 31) | 0.039402s |
| event_bus/registrations.23.rb | EventBus::Registrations | LOI        | killed    | 8 (of 31)  | 0.038908s |
| event_bus/registrations.24.rb | EventBus::Registrations | LOI        | killed    | 9 (of 31)  | 0.030461s |
| event_bus/registrations.25.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.039189s |
| event_bus/registrations.26.rb | EventBus::Registrations | LOI        | killed    | 13 (of 31) | 0.051285s |
| event_bus/registrations.27.rb | EventBus::Registrations | LOI        | killed    | 1 (of 31)  | 0.05042s  |
| event_bus/registrations.28.rb | EventBus::Registrations | LOI        | killed    | 10 (of 31) | 0.07343s  |
| event_bus/registrations.29.rb | EventBus::Registrations | LOI        | killed    | 19 (of 31) | 0.045765s |
| event_bus/registrations.30.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.028747s |
| event_bus/registrations.31.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.021559s |
| event_bus/registrations.32.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.022631s |
| event_bus/registrations.33.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.028901s |
| event_bus/registrations.34.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.031961s |
| event_bus/registrations.35.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.020941s |
| event_bus/registrations.36.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.020681s |
| event_bus/registrations.37.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.027151s |
| event_bus/registrations.38.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.028807s |
| event_bus/registrations.39.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.020968s |
| event_bus/registrations.40.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.028554s |
| event_bus/registrations.41.rb | EventBus::Registrations | SAOR       | killed    | 1 (of 31)  | 0.03014s  |
```

```
| event_bus/registrations.42.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.020533s |
| event_bus/registrations.43.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.026158s |
| event_bus/registrations.44.rb | EventBus::Registrations | UAOI       | killed    | 10 (of 31) | 0.038609s |
| event_bus/registrations.45.rb | EventBus::Registrations | UAOI       | stillborn | n/a        | n/a       |
| event_bus/registrations.46.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.026709s |
| event_bus/registrations.47.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.026476s |
| event_bus/registrations.48.rb | EventBus::Registrations | UAOI       | killed    | 9 (of 31)  | 0.032533s |
| event_bus/registrations.49.rb | EventBus::Registrations | UAOI       | killed    | 10 (of 31) | 0.040806s |
| event_bus/registrations.50.rb | EventBus::Registrations | UAOI       | killed    | 8 (of 31)  | 0.029494s |
| event_bus/registrations.51.rb | EventBus::Registrations | UAOI       | killed    | 9 (of 31)  | 0.040073s |
| event_bus/registrations.52.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.04766s  |
| event_bus/registrations.53.rb | EventBus::Registrations | UAOI       | killed    | 13 (of 31) | 0.045031s |
| event_bus/registrations.54.rb | EventBus::Registrations | UAOI       | killed    | 1 (of 31)  | 0.038901s |
| event_bus/registrations.55.rb | EventBus::Registrations | UAOI       | killed    | 10 (of 31) | 0.063441s |
| event_bus/registrations.56.rb | EventBus::Registrations | UAOI       | killed    | 19 (of 31) | 0.056299s |
+-------------------------------+-------------------------+------------+-----------+------------+-----------+
```

full_name

```
$ multimutiny -ci -cit 90

--------------------------------------------------------------------------------
                              MultiMutiny Report
--------------------------------------------------------------------------------

                               CI BUILD PASSED

+----------------------------+------+
| # Mutants Created          | 51   |
| # Mutants Killed           | 51   |
| Pessimistic Mutation Score | 100% |
| CI Threshold               | 90%  |
+----------------------------+------+

+-----------------+----------+------------+----------+------------+-----------+
| Mutant          | Target   | Operator   | Status   | # Tests    | Time      |
|-----------------+----------+------------+----------+------------+-----------|
| full_name.0.rb  | FullName | COI        | killed   | 3 (of 15)  | 0.04193s  |
| full_name.1.rb  | FullName | COI        | killed   | 15 (of 15) | 0.042936s |
| full_name.2.rb  | FullName | COI        | killed   | 3 (of 15)  | 0.044003s |
| full_name.3.rb  | FullName | COI        | killed   | 6 (of 15)  | 0.039877s |
| full_name.4.rb  | FullName | COI        | killed   | 10 (of 15) | 0.040465s |
| full_name.5.rb  | FullName | COI        | killed   | 1 (of 15)  | 0.041161s |
| full_name.6.rb  | FullName | COI        | killed   | 1 (of 15)  | 0.038068s |
| full_name.7.rb  | FullName | COI        | killed   | 6 (of 15)  | 0.036082s |
| full_name.8.rb  | FullName | COI        | killed   | 6 (of 15)  | 0.03985s  |
| full_name.9.rb  | FullName | COI        | killed   | 1 (of 15)  | 0.035455s |
| full_name.10.rb | FullName | COI        | killed   | 1 (of 15)  | 0.038636s |
| full_name.11.rb | FullName | COI        | killed   | 1 (of 15)  | 0.038325s |
| full_name.12.rb | FullName | COI        | killed   | 1 (of 15)  | 0.039776s |
| full_name.13.rb | FullName | COI        | killed   | 11 (of 15) | 0.046286s |
| full_name.14.rb | FullName | COI        | killed   | 11 (of 15) | 0.039437s |
| full_name.15.rb | FullName | COR        | killed   | 8 (of 15)  | 0.027894s |
| full_name.16.rb | FullName | COR        | killed   | 6 (of 15)  | 0.040467s |
| full_name.17.rb | FullName | COR        | killed   | 9 (of 15)  | 0.032095s |
| full_name.18.rb | FullName | COR        | killed   | 9 (of 15)  | 0.027813s |
| full_name.19.rb | FullName | COR        | killed   | 5 (of 15)  | 0.027666s |
| full_name.20.rb | FullName | COR        | killed   | 3 (of 15)  | 0.042332s |
| full_name.21.rb | FullName | LOI        | killed   | 3 (of 15)  | 0.027737s |
| full_name.22.rb | FullName | LOI        | killed   | 15 (of 15) | 0.028771s |
| full_name.23.rb | FullName | LOI        | killed   | 3 (of 15)  | 0.028366s |
| full_name.24.rb | FullName | LOI        | killed   | 6 (of 15)  | 0.029405s |
| full_name.25.rb | FullName | LOI        | killed   | 10 (of 15) | 0.029342s |
| full_name.26.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.026229s |
| full_name.27.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.028093s |
| full_name.28.rb | FullName | LOI        | killed   | 6 (of 15)  | 0.030884s |
| full_name.29.rb | FullName | LOI        | killed   | 6 (of 15)  | 0.027865s |
| full_name.30.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.026223s |
| full_name.31.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.027676s |
| full_name.32.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.026059s |
| full_name.33.rb | FullName | LOI        | killed   | 1 (of 15)  | 0.026222s |
| full_name.34.rb | FullName | LOI        | killed   | 11 (of 15) | 0.029065s |
| full_name.35.rb | FullName | LOI        | killed   | 11 (of 15) | 0.032234s |
| full_name.36.rb | FullName | UAOI       | killed   | 3 (of 15)  | 0.027367s |
```

```
| full_name.37.rb | FullName | UAOI       | killed   | 15 (of 15) | 0.028659s |
| full_name.38.rb | FullName | UAOI       | killed   | 3 (of 15)  | 0.026948s |
| full_name.39.rb | FullName | UAOI       | killed   | 6 (of 15)  | 0.028032s |
| full_name.40.rb | FullName | UAOI       | killed   | 10 (of 15) | 0.027033s |
| full_name.41.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.028182s |
| full_name.42.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.028905s |
| full_name.43.rb | FullName | UAOI       | killed   | 6 (of 15)  | 0.028302s |
| full_name.44.rb | FullName | UAOI       | killed   | 6 (of 15)  | 0.027506s |
| full_name.45.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.027445s |
| full_name.46.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.026962s |
| full_name.47.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.026493s |
| full_name.48.rb | FullName | UAOI       | killed   | 1 (of 15)  | 0.02702s  |
| full_name.49.rb | FullName | UAOI       | killed   | 11 (of 15) | 0.029148s |
| full_name.50.rb | FullName | UAOI       | killed   | 11 (of 15) | 0.02864s  |
+-----------------+----------+------------+----------+------------+-----------+
```

lumberjack

The column giving target classes has been omitted from this report in order to enable it to fit on the page. The target class for a mutant can be inferred from the mutant's filename quite easily though: For the first mutant, for example, the filename of `lumberjack/device/date_rolling_log_file.0.rb` translates to a target class of `Lumberjack::Device::DateRollingLogFile`.

```
$ multimutiny -ci -cit 90

-----------------------------------------------------------------------------------------------------
                                        MultiMutiny Report
-----------------------------------------------------------------------------------------------------

                                        CI BUILD FAILED

+----------------------------+-----+
| # Mutants Created          | 565 |
| # Mutants Killed           | 480 |
| Pessimistic Mutation Score | 85% |
| CI Threshold               | 90% |
+----------------------------+-----+

+-----------------------------------------------+------------+-----------+------------+-----------+
| Mutant                                        | Operator   | Status    | # Tests    | Time      |
|-----------------------------------------------+------------+-----------+------------+-----------|
| lumberjack/device/date_rolling_log_file.0.rb  | COI        | survived  | 8 (of 8)   | 0.011754s |
| lumberjack/device/date_rolling_log_file.1.rb  | COI        | survived  | 8 (of 8)   | 0.012023s |
| lumberjack/device/date_rolling_log_file.2.rb  | COI        | killed    | 1 (of 8)   | 0.030989s |
| lumberjack/device/date_rolling_log_file.3.rb  | COI        | killed    | 2 (of 8)   | 0.033941s |
| lumberjack/device/date_rolling_log_file.4.rb  | COI        | killed    | 3 (of 8)   | 0.03235s  |
| lumberjack/device/date_rolling_log_file.5.rb  | COI        | survived  | 8 (of 8)   | 0.011397s |
| lumberjack/device/date_rolling_log_file.6.rb  | COR        | killed    | 1 (of 8)   | 0.031201s |
| lumberjack/device/date_rolling_log_file.7.rb  | COR        | killed    | 1 (of 8)   | 0.029981s |
| lumberjack/device/date_rolling_log_file.8.rb  | COR        | survived  | 8 (of 8)   | 0.010744s |
| lumberjack/device/date_rolling_log_file.9.rb  | COR        | survived  | 8 (of 8)   | 0.013716s |
| lumberjack/device/date_rolling_log_file.10.rb | COR        | survived  | 8 (of 8)   | 0.012176s |
| lumberjack/device/date_rolling_log_file.11.rb | COR        | survived  | 8 (of 8)   | 0.011449s |
| lumberjack/device/date_rolling_log_file.12.rb | COR        | survived  | 8 (of 8)   | 0.01272s  |
| lumberjack/device/date_rolling_log_file.13.rb | COR        | survived  | 8 (of 8)   | 0.012283s |
| lumberjack/device/date_rolling_log_file.14.rb | RER        | survived  | 8 (of 8)   | 0.012564s |
| lumberjack/device/date_rolling_log_file.15.rb | RER        | survived  | 8 (of 8)   | 0.012903s |
| lumberjack/device/date_rolling_log_file.16.rb | RER        | survived  | 8 (of 8)   | 0.012393s |
| lumberjack/device/date_rolling_log_file.17.rb | RER        | killed    | 1 (of 8)   | 0.029008s |
| lumberjack/device/date_rolling_log_file.18.rb | RER        | survived  | 8 (of 8)   | 0.011037s |
| lumberjack/device/date_rolling_log_file.19.rb | RER        | killed    | 1 (of 8)   | 0.030551s |
| lumberjack/device/date_rolling_log_file.20.rb | RER        | survived  | 8 (of 8)   | 0.012608s |
| lumberjack/device/date_rolling_log_file.21.rb | RER        | killed    | 2 (of 8)   | 0.029524s |
| lumberjack/device/date_rolling_log_file.22.rb | RER        | survived  | 8 (of 8)   | 0.011419s |
| lumberjack/device/date_rolling_log_file.23.rb | RER        | killed    | 2 (of 8)   | 0.033524s |
| lumberjack/device/date_rolling_log_file.24.rb | RER        | survived  | 8 (of 8)   | 0.012445s |
| lumberjack/device/date_rolling_log_file.25.rb | RER        | killed    | 3 (of 8)   | 0.031929s |
| lumberjack/device/date_rolling_log_file.26.rb | RER        | survived  | 8 (of 8)   | 0.011679s |
| lumberjack/device/date_rolling_log_file.27.rb | RER        | killed    | 3 (of 8)   | 0.036239s |
| lumberjack/device/date_rolling_log_file.28.rb | ROR        | survived  | 8 (of 8)   | 0.012178s |
| lumberjack/device/date_rolling_log_file.29.rb | ROR        | survived  | 8 (of 8)   | 0.01206s  |
| lumberjack/device/date_rolling_log_file.30.rb | ROR        | survived  | 8 (of 8)   | 0.014473s |
```

lumberjack/device/date_rolling_log_file.31.rb	ROR	survived	8 (of 8)	0.011013s
lumberjack/device/date_rolling_log_file.32.rb	ROR	survived	8 (of 8)	0.011159s
lumberjack/device/date_rolling_log_file.33.rb	ROR	killed	1 (of 8)	0.028348s
lumberjack/device/date_rolling_log_file.34.rb	ROR	survived	8 (of 8)	0.012714s
lumberjack/device/date_rolling_log_file.35.rb	ROR	killed	1 (of 8)	0.028525s
lumberjack/device/date_rolling_log_file.36.rb	ROR	survived	8 (of 8)	0.010954s
lumberjack/device/date_rolling_log_file.37.rb	ROR	killed	1 (of 8)	0.028647s
lumberjack/device/date_rolling_log_file.38.rb	ROR	killed	1 (of 8)	0.028386s
lumberjack/device/date_rolling_log_file.39.rb	ROR	survived	8 (of 8)	0.01102s
lumberjack/device/date_rolling_log_file.40.rb	ROR	survived	8 (of 8)	0.011178s
lumberjack/device/date_rolling_log_file.41.rb	ROR	survived	8 (of 8)	0.012048s
lumberjack/device/date_rolling_log_file.42.rb	ROR	survived	8 (of 8)	0.011108s
lumberjack/device/date_rolling_log_file.43.rb	ROR	killed	2 (of 8)	0.029771s
lumberjack/device/date_rolling_log_file.44.rb	ROR	survived	8 (of 8)	0.011918s
lumberjack/device/date_rolling_log_file.45.rb	ROR	killed	2 (of 8)	0.035335s
lumberjack/device/date_rolling_log_file.46.rb	ROR	survived	8 (of 8)	0.01142s
lumberjack/device/date_rolling_log_file.47.rb	ROR	killed	2 (of 8)	0.032267s
lumberjack/device/date_rolling_log_file.48.rb	ROR	killed	2 (of 8)	0.031954s
lumberjack/device/date_rolling_log_file.49.rb	ROR	survived	8 (of 8)	0.012044s
lumberjack/device/date_rolling_log_file.50.rb	ROR	survived	8 (of 8)	0.011201s
lumberjack/device/date_rolling_log_file.51.rb	ROR	survived	8 (of 8)	0.014187s
lumberjack/device/date_rolling_log_file.52.rb	ROR	survived	8 (of 8)	0.011626s
lumberjack/device/date_rolling_log_file.53.rb	ROR	killed	3 (of 8)	0.033565s
lumberjack/device/date_rolling_log_file.54.rb	ROR	survived	8 (of 8)	0.012209s
lumberjack/device/date_rolling_log_file.55.rb	ROR	killed	3 (of 8)	0.034917s
lumberjack/device/date_rolling_log_file.56.rb	ROR	survived	8 (of 8)	0.011415s
lumberjack/device/date_rolling_log_file.57.rb	ROR	killed	3 (of 8)	0.034388s
lumberjack/device/date_rolling_log_file.58.rb	ROR	killed	3 (of 8)	0.03426s
lumberjack/device/date_rolling_log_file.59.rb	ROR	survived	8 (of 8)	0.014418s
lumberjack/device/date_rolling_log_file.60.rb	ROR	survived	8 (of 8)	0.011477s
lumberjack/device/date_rolling_log_file.61.rb	ROR	survived	8 (of 8)	0.011368s
lumberjack/device/date_rolling_log_file.62.rb	ROR	survived	8 (of 8)	0.011526s
lumberjack/device/date_rolling_log_file.63.rb	LOI	killed	1 (of 8)	0.028929s
lumberjack/device/rolling_log_file.0.rb	COD	killed	1 (of 34)	0.028067s
lumberjack/device/rolling_log_file.1.rb	COI	killed	1 (of 34)	0.031785s
lumberjack/device/rolling_log_file.2.rb	COI	killed	1 (of 34)	0.031874s
lumberjack/device/rolling_log_file.3.rb	COI	survived	34 (of 34)	1.080569s
lumberjack/device/rolling_log_file.4.rb	COI	survived	34 (of 34)	0.569646s
lumberjack/device/rolling_log_file.5.rb	COI	killed	1 (of 34)	0.288139s
lumberjack/device/rolling_log_file.6.rb	COI	killed	1 (of 34)	0.097981s
lumberjack/device/rolling_log_file.7.rb	COI	survived	34 (of 34)	0.76325s
lumberjack/device/rolling_log_file.8.rb	COI	survived	34 (of 34)	0.580644s
lumberjack/device/rolling_log_file.9.rb	COI	killed	1 (of 34)	0.046214s
lumberjack/device/rolling_log_file.10.rb	COI	survived	34 (of 34)	0.537736s
lumberjack/device/rolling_log_file.11.rb	COI	killed	1 (of 34)	0.201654s
lumberjack/device/rolling_log_file.12.rb	COI	killed	1 (of 34)	0.02852s
lumberjack/device/rolling_log_file.13.rb	COI	killed	1 (of 34)	0.02708s
lumberjack/device/rolling_log_file.14.rb	COI	killed	7 (of 34)	0.112444s
lumberjack/device/rolling_log_file.15.rb	COI	survived	34 (of 34)	0.925635s
lumberjack/device/rolling_log_file.16.rb	COI	killed	1 (of 34)	0.168414s
lumberjack/device/rolling_log_file.17.rb	COI	killed	1 (of 34)	0.067013s
lumberjack/device/rolling_log_file.18.rb	COI	killed	1 (of 34)	0.172034s
lumberjack/device/rolling_log_file.19.rb	COI	killed	1 (of 34)	0.030261s
lumberjack/device/rolling_log_file.20.rb	COI	killed	1 (of 34)	0.02662s
lumberjack/device/rolling_log_file.21.rb	COI	killed	1 (of 34)	0.027866s
lumberjack/device/rolling_log_file.22.rb	COI	survived	34 (of 34)	0.543692s
lumberjack/device/rolling_log_file.23.rb	COI	killed	1 (of 34)	0.036672s
lumberjack/device/rolling_log_file.24.rb	COI	killed	1 (of 34)	0.380485s
lumberjack/device/rolling_log_file.25.rb	COI	killed	11 (of 34)	0.770433s
lumberjack/device/rolling_log_file.26.rb	COR	killed	1 (of 34)	0.132685s
lumberjack/device/rolling_log_file.27.rb	COR	killed	1 (of 34)	0.033743s
lumberjack/device/rolling_log_file.28.rb	COR	killed	1 (of 34)	0.027331s
lumberjack/device/rolling_log_file.29.rb	COR	survived	34 (of 34)	0.471533s
lumberjack/device/rolling_log_file.30.rb	COR	killed	1 (of 34)	0.032238s
lumberjack/device/rolling_log_file.31.rb	COR	killed	1 (of 34)	0.135003s
lumberjack/device/rolling_log_file.32.rb	COR	killed	1 (of 34)	0.264796s
lumberjack/device/rolling_log_file.33.rb	COR	killed	1 (of 34)	0.060618s
lumberjack/device/rolling_log_file.34.rb	COR	killed	1 (of 34)	0.074715s
lumberjack/device/rolling_log_file.35.rb	COR	killed	1 (of 34)	0.026762s
lumberjack/device/rolling_log_file.36.rb	RER	killed	11 (of 34)	0.664908s
lumberjack/device/rolling_log_file.37.rb	RER	killed	11 (of 34)	0.927851s
lumberjack/device/rolling_log_file.38.rb	RER	killed	1 (of 34)	0.029861s
lumberjack/device/rolling_log_file.39.rb	RER	killed	1 (of 34)	0.027136s
lumberjack/device/rolling_log_file.40.rb	RER	killed	1 (of 34)	0.057392s
lumberjack/device/rolling_log_file.41.rb	RER	killed	1 (of 34)	0.033129s

lumberjack/device/rolling_log_file.42.rb	RER	killed	1 (of 34)	0.046486s
lumberjack/device/rolling_log_file.43.rb	RER	killed	1 (of 34)	0.030356s
lumberjack/device/rolling_log_file.44.rb	RER	survived	34 (of 34)	0.646037s
lumberjack/device/rolling_log_file.45.rb	RER	killed	1 (of 34)	0.131319s
lumberjack/device/rolling_log_file.46.rb	ROR	killed	1 (of 34)	0.04677s
lumberjack/device/rolling_log_file.47.rb	ROR	killed	11 (of 34)	0.401994s
lumberjack/device/rolling_log_file.48.rb	ROR	killed	11 (of 34)	0.794366s
lumberjack/device/rolling_log_file.49.rb	ROR	killed	1 (of 34)	0.037141s
lumberjack/device/rolling_log_file.50.rb	ROR	survived	34 (of 34)	0.588223s
lumberjack/device/rolling_log_file.51.rb	ROR	killed	9 (of 34)	0.065204s
lumberjack/device/rolling_log_file.52.rb	ROR	killed	1 (of 34)	0.032888s
lumberjack/device/rolling_log_file.53.rb	ROR	killed	1 (of 34)	0.051102s
lumberjack/device/rolling_log_file.54.rb	ROR	killed	1 (of 34)	0.028958s
lumberjack/device/rolling_log_file.55.rb	ROR	killed	9 (of 34)	0.068774s
lumberjack/device/rolling_log_file.56.rb	ROR	killed	1 (of 34)	0.034955s
lumberjack/device/rolling_log_file.57.rb	ROR	killed	1 (of 34)	0.261927s
lumberjack/device/rolling_log_file.58.rb	ROR	killed	1 (of 34)	0.046212s
lumberjack/device/rolling_log_file.59.rb	ROR	survived	34 (of 34)	0.784842s
lumberjack/device/rolling_log_file.60.rb	ROR	killed	1 (of 34)	0.050484s
lumberjack/device/rolling_log_file.61.rb	ROR	killed	1 (of 34)	0.059133s
lumberjack/device/rolling_log_file.62.rb	ROR	survived	34 (of 34)	0.603923s
lumberjack/device/rolling_log_file.63.rb	ROR	killed	1 (of 34)	0.031285s
lumberjack/device/rolling_log_file.64.rb	ROR	killed	1 (of 34)	0.044876s
lumberjack/device/rolling_log_file.65.rb	ROR	killed	1 (of 34)	0.027427s
lumberjack/device/rolling_log_file.66.rb	ROR	killed	1 (of 34)	0.047251s
lumberjack/device/rolling_log_file.67.rb	ROR	killed	1 (of 34)	0.033048s
lumberjack/device/rolling_log_file.68.rb	ROR	killed	1 (of 34)	0.044292s
lumberjack/device/rolling_log_file.69.rb	ROR	survived	34 (of 34)	0.747725s
lumberjack/device/rolling_log_file.70.rb	ROR	killed	1 (of 34)	0.151335s
lumberjack/device/rolling_log_file.71.rb	LOI	killed	1 (of 34)	0.046189s
lumberjack/device/rolling_log_file.72.rb	LOI	killed	1 (of 34)	0.045228s
lumberjack/device/rolling_log_file.73.rb	LOI	killed	2 (of 34)	0.050558s
lumberjack/device/rolling_log_file.74.rb	LOI	killed	1 (of 34)	0.049578s
lumberjack/device/rolling_log_file.75.rb	LOI	killed	1 (of 34)	0.142677s
lumberjack/device/rolling_log_file.76.rb	LOI	survived	34 (of 34)	0.389625s
lumberjack/device/rolling_log_file.77.rb	LOI	killed	9 (of 34)	0.063395s
lumberjack/device/rolling_log_file.78.rb	LOI	killed	1 (of 34)	0.033276s
lumberjack/device/rolling_log_file.79.rb	LOI	killed	1 (of 34)	0.047845s
lumberjack/device/rolling_log_file.80.rb	LOI	killed	1 (of 34)	0.057987s
lumberjack/device/rolling_log_file.81.rb	LOI	killed	1 (of 34)	0.032857s
lumberjack/device/rolling_log_file.82.rb	LOI	killed	1 (of 34)	0.054717s
lumberjack/device/rolling_log_file.83.rb	LOI	killed	7 (of 34)	0.057101s
lumberjack/device/rolling_log_file.84.rb	LOI	survived	34 (of 34)	0.684121s
lumberjack/device/rolling_log_file.85.rb	LOI	killed	1 (of 34)	0.033315s
lumberjack/device/rolling_log_file.86.rb	LOI	killed	1 (of 34)	0.043402s
lumberjack/device/rolling_log_file.87.rb	LOI	killed	1 (of 34)	0.1022s
lumberjack/device/rolling_log_file.88.rb	LOI	killed	1 (of 34)	0.046027s
lumberjack/device/rolling_log_file.89.rb	LOI	killed	1 (of 34)	0.034126s
lumberjack/device/rolling_log_file.90.rb	LOI	killed	1 (of 34)	0.028292s
lumberjack/device/rolling_log_file.91.rb	LOI	killed	1 (of 34)	0.048014s
lumberjack/device/rolling_log_file.92.rb	LOI	killed	1 (of 34)	0.032865s
lumberjack/device/rolling_log_file.93.rb	LOI	killed	1 (of 34)	0.047379s
lumberjack/device/rolling_log_file.94.rb	LOI	killed	1 (of 34)	0.031602s
lumberjack/device/rolling_log_file.95.rb	LOI	killed	11 (of 34)	0.720951s
lumberjack/device/rolling_log_file.96.rb	UAOI	killed	1 (of 34)	0.033804s
lumberjack/device/rolling_log_file.97.rb	UAOI	killed	1 (of 34)	0.211377s
lumberjack/device/rolling_log_file.98.rb	UAOI	killed	2 (of 34)	0.033097s
lumberjack/device/rolling_log_file.99.rb	UAOI	killed	11 (of 34)	0.623285s
lumberjack/device/rolling_log_file.100.rb	UAOI	killed	1 (of 34)	0.029049s
lumberjack/device/rolling_log_file.101.rb	UAOI	killed	1 (of 34)	0.028131s
lumberjack/device/rolling_log_file.102.rb	UAOI	killed	1 (of 34)	0.028202s
lumberjack/device/rolling_log_file.103.rb	UAOI	killed	1 (of 34)	0.027344s
lumberjack/device/rolling_log_file.104.rb	UAOI	killed	1 (of 34)	0.027927s
lumberjack/device/rolling_log_file.105.rb	UAOI	killed	1 (of 34)	0.06907s
lumberjack/device/rolling_log_file.106.rb	UAOI	killed	1 (of 34)	0.087839s
lumberjack/device/rolling_log_file.107.rb	UAOI	killed	11 (of 34)	0.486091s
lumberjack/device/rolling_log_file.108.rb	UAOI	killed	7 (of 34)	0.198258s
lumberjack/device/rolling_log_file.109.rb	UAOI	killed	1 (of 34)	0.029623s
lumberjack/device/rolling_log_file.110.rb	UAOI	killed	1 (of 34)	0.026831s
lumberjack/device/rolling_log_file.111.rb	UAOI	killed	11 (of 34)	0.531363s
lumberjack/device/rolling_log_file.112.rb	UAOI	killed	1 (of 34)	0.028937s
lumberjack/device/rolling_log_file.113.rb	UAOI	killed	1 (of 34)	0.029452s
lumberjack/device/rolling_log_file.114.rb	UAOI	killed	1 (of 34)	0.027645s
lumberjack/device/rolling_log_file.115.rb	UAOI	killed	1 (of 34)	0.032295s
lumberjack/device/rolling_log_file.116.rb	UAOI	killed	1 (of 34)	0.028414s

lumberjack/device/rolling_log_file.117.rb	UAOI	killed	1 (of 34)	0.029404s
lumberjack/device/rolling_log_file.118.rb	UAOI	killed	1 (of 34)	0.029535s
lumberjack/device/rolling_log_file.119.rb	UAOI	killed	11 (of 34)	0.812782s
lumberjack/device/size_rolling_log_file.0.rb	BAOR	killed	1 (of 10)	0.062865s
lumberjack/device/size_rolling_log_file.1.rb	BAOR	killed	1 (of 10)	0.042104s
lumberjack/device/size_rolling_log_file.2.rb	BAOR	killed	1 (of 10)	0.030696s
lumberjack/device/size_rolling_log_file.3.rb	BAOR	killed	1 (of 10)	0.043871s
lumberjack/device/size_rolling_log_file.4.rb	COI	killed	1 (of 10)	0.028808s
lumberjack/device/size_rolling_log_file.5.rb	COI	killed	1 (of 10)	0.150623s
lumberjack/device/size_rolling_log_file.6.rb	COI	killed	1 (of 10)	0.031802s
lumberjack/device/size_rolling_log_file.7.rb	COI	killed	5 (of 10)	0.06088s
lumberjack/device/size_rolling_log_file.8.rb	COR	survived	10 (of 10)	0.014054s
lumberjack/device/size_rolling_log_file.9.rb	COR	killed	1 (of 10)	0.02919s
lumberjack/device/size_rolling_log_file.10.rb	RER	killed	1 (of 10)	0.083682s
lumberjack/device/size_rolling_log_file.11.rb	RER	killed	1 (of 10)	0.046216s
lumberjack/device/size_rolling_log_file.12.rb	RER	killed	5 (of 10)	0.046881s
lumberjack/device/size_rolling_log_file.13.rb	RER	killed	1 (of 10)	0.052836s
lumberjack/device/size_rolling_log_file.14.rb	ROR	killed	1 (of 10)	0.027755s
lumberjack/device/size_rolling_log_file.15.rb	ROR	killed	1 (of 10)	0.057376s
lumberjack/device/size_rolling_log_file.16.rb	ROR	killed	1 (of 10)	0.030292s
lumberjack/device/size_rolling_log_file.17.rb	ROR	killed	1 (of 10)	0.053078s
lumberjack/device/size_rolling_log_file.18.rb	ROR	survived	10 (of 10)	0.01222s
lumberjack/device/size_rolling_log_file.19.rb	ROR	killed	5 (of 10)	0.053762s
lumberjack/device/size_rolling_log_file.20.rb	ROR	killed	1 (of 10)	0.054372s
lumberjack/device/size_rolling_log_file.21.rb	ROR	killed	5 (of 10)	0.048115s
lumberjack/device/size_rolling_log_file.22.rb	ROR	killed	5 (of 10)	0.057264s
lumberjack/device/size_rolling_log_file.23.rb	ROR	survived	10 (of 10)	0.024262s
lumberjack/device/size_rolling_log_file.24.rb	LOI	killed	1 (of 10)	0.050876s
lumberjack/device/size_rolling_log_file.25.rb	LOI	killed	1 (of 10)	0.027266s
lumberjack/device/size_rolling_log_file.26.rb	LOI	killed	1 (of 10)	0.046582s
lumberjack/device/size_rolling_log_file.27.rb	LOI	killed	1 (of 10)	0.030728s
lumberjack/device/size_rolling_log_file.28.rb	LOI	killed	1 (of 10)	0.074993s
lumberjack/device/size_rolling_log_file.29.rb	LOI	killed	2 (of 10)	0.044646s
lumberjack/device/size_rolling_log_file.30.rb	LOI	killed	2 (of 10)	0.26777s
lumberjack/device/size_rolling_log_file.31.rb	LOI	killed	1 (of 10)	0.059144s
lumberjack/device/size_rolling_log_file.32.rb	LOI	killed	3 (of 10)	0.059672s
lumberjack/device/size_rolling_log_file.33.rb	LOI	killed	3 (of 10)	0.054717s
lumberjack/device/size_rolling_log_file.34.rb	LOI	killed	4 (of 10)	0.043262s
lumberjack/device/size_rolling_log_file.35.rb	LOI	killed	1 (of 10)	0.059525s
lumberjack/device/size_rolling_log_file.36.rb	SAOR	killed	2 (of 10)	0.046202s
lumberjack/device/size_rolling_log_file.37.rb	SAOR	killed	2 (of 10)	0.057424s
lumberjack/device/size_rolling_log_file.38.rb	SAOR	killed	2 (of 10)	0.043147s
lumberjack/device/size_rolling_log_file.39.rb	SAOR	killed	1 (of 10)	0.058103s
lumberjack/device/size_rolling_log_file.40.rb	SAOR	killed	2 (of 10)	0.045296s
lumberjack/device/size_rolling_log_file.41.rb	SAOR	killed	2 (of 10)	0.123233s
lumberjack/device/size_rolling_log_file.42.rb	SAOR	killed	1 (of 10)	0.055991s
lumberjack/device/size_rolling_log_file.43.rb	SAOR	killed	2 (of 10)	0.033419s
lumberjack/device/size_rolling_log_file.44.rb	SAOR	killed	2 (of 10)	0.043839s
lumberjack/device/size_rolling_log_file.45.rb	SAOR	killed	2 (of 10)	0.03675s
lumberjack/device/size_rolling_log_file.46.rb	SAOR	killed	1 (of 10)	0.149449s
lumberjack/device/size_rolling_log_file.47.rb	SAOR	killed	2 (of 10)	0.044978s
lumberjack/device/size_rolling_log_file.48.rb	SAOR	killed	3 (of 10)	0.054879s
lumberjack/device/size_rolling_log_file.49.rb	SAOR	killed	3 (of 10)	0.05065s
lumberjack/device/size_rolling_log_file.50.rb	SAOR	killed	3 (of 10)	0.237451s
lumberjack/device/size_rolling_log_file.51.rb	SAOR	killed	3 (of 10)	0.053401s
lumberjack/device/size_rolling_log_file.52.rb	SAOR	killed	3 (of 10)	0.036828s
lumberjack/device/size_rolling_log_file.53.rb	SAOR	killed	1 (of 10)	0.058657s
lumberjack/device/size_rolling_log_file.54.rb	SAOR	killed	3 (of 10)	0.038537s
lumberjack/device/size_rolling_log_file.55.rb	SAOR	killed	3 (of 10)	0.049818s
lumberjack/device/size_rolling_log_file.56.rb	SAOR	killed	3 (of 10)	0.035811s
lumberjack/device/size_rolling_log_file.57.rb	SAOR	killed	1 (of 10)	0.059514s
lumberjack/device/size_rolling_log_file.58.rb	SAOR	killed	3 (of 10)	0.146324s
lumberjack/device/size_rolling_log_file.59.rb	SAOR	killed	3 (of 10)	0.058697s
lumberjack/device/size_rolling_log_file.60.rb	SAOR	killed	1 (of 10)	0.057007s
lumberjack/device/size_rolling_log_file.61.rb	SAOR	killed	4 (of 10)	0.045271s
lumberjack/device/size_rolling_log_file.62.rb	SAOR	killed	4 (of 10)	0.054083s
lumberjack/device/size_rolling_log_file.63.rb	SAOR	killed	4 (of 10)	0.266155s
lumberjack/device/size_rolling_log_file.64.rb	SAOR	killed	1 (of 10)	0.06644s
lumberjack/device/size_rolling_log_file.65.rb	SAOR	killed	4 (of 10)	0.040116s
lumberjack/device/size_rolling_log_file.66.rb	SAOR	killed	4 (of 10)	0.050931s
lumberjack/device/size_rolling_log_file.67.rb	SAOR	killed	4 (of 10)	0.032189s
lumberjack/device/size_rolling_log_file.68.rb	SAOR	killed	1 (of 10)	0.057218s
lumberjack/device/size_rolling_log_file.69.rb	SAOR	killed	4 (of 10)	0.03812s
lumberjack/device/size_rolling_log_file.70.rb	SAOR	survived	10 (of 10)	0.013946s
lumberjack/device/size_rolling_log_file.71.rb	SAOR	killed	1 (of 10)	0.120233s

lumberjack/device/size_rolling_log_file.72.rb	UAOI	killed	1 (of 10)	0.029876s
lumberjack/device/size_rolling_log_file.73.rb	UAOI	killed	1 (of 10)	0.048737s
lumberjack/device/size_rolling_log_file.74.rb	UAOI	killed	1 (of 10)	0.027897s
lumberjack/device/size_rolling_log_file.75.rb	UAOI	killed	1 (of 10)	0.04633s
lumberjack/device/size_rolling_log_file.76.rb	UAOI	killed	2 (of 10)	0.045797s
lumberjack/device/size_rolling_log_file.77.rb	UAOI	killed	2 (of 10)	0.056472s
lumberjack/device/size_rolling_log_file.78.rb	UAOI	killed	2 (of 10)	0.077556s
lumberjack/device/size_rolling_log_file.79.rb	UAOI	killed	1 (of 10)	0.059085s
lumberjack/device/size_rolling_log_file.80.rb	UAOI	killed	3 (of 10)	0.057336s
lumberjack/device/size_rolling_log_file.81.rb	UAOI	killed	4 (of 10)	0.040072s
lumberjack/device/size_rolling_log_file.82.rb	UAOI	killed	1 (of 10)	0.056949s
lumberjack/device/writer.0.rb	BAOR	killed	10 (of 17)	0.058585s
lumberjack/device/writer.1.rb	BAOR	killed	10 (of 17)	0.042514s
lumberjack/device/writer.2.rb	BAOR	killed	10 (of 17)	0.058177s
lumberjack/device/writer.3.rb	BAOR	survived	17 (of 17)	0.016218s
lumberjack/device/writer.4.rb	COI	survived	17 (of 17)	0.01278s
lumberjack/device/writer.5.rb	COI	survived	17 (of 17)	0.017756s
lumberjack/device/writer.6.rb	COI	survived	17 (of 17)	0.015648s
lumberjack/device/writer.7.rb	COI	survived	17 (of 17)	0.01313s
lumberjack/device/writer.8.rb	COI	survived	17 (of 17)	0.01406s
lumberjack/device/writer.9.rb	COI	killed	3 (of 17)	0.064838s
lumberjack/device/writer.10.rb	COI	killed	1 (of 17)	0.049969s
lumberjack/device/writer.11.rb	COI	survived	17 (of 17)	0.014963s
lumberjack/device/writer.12.rb	COI	survived	17 (of 17)	0.013774s
lumberjack/device/writer.13.rb	COR	killed	5 (of 17)	0.051729s
lumberjack/device/writer.14.rb	COR	killed	1 (of 17)	0.06152s
lumberjack/device/writer.15.rb	COR	killed	1 (of 17)	0.039506s
lumberjack/device/writer.16.rb	COR	killed	1 (of 17)	0.061859s
lumberjack/device/writer.17.rb	COR	killed	1 (of 17)	0.02631s
lumberjack/device/writer.18.rb	COR	killed	11 (of 17)	0.071694s
lumberjack/device/writer.19.rb	RER	killed	1 (of 17)	0.051083s
lumberjack/device/writer.20.rb	RER	killed	2 (of 17)	0.051364s
lumberjack/device/writer.21.rb	ROR	killed	1 (of 17)	0.063878s
lumberjack/device/writer.22.rb	ROR	killed	1 (of 17)	0.048682s
lumberjack/device/writer.23.rb	ROR	killed	2 (of 17)	0.064174s
lumberjack/device/writer.24.rb	ROR	killed	1 (of 17)	0.129283s
lumberjack/device/writer.25.rb	ROR	survived	17 (of 17)	0.013781s
lumberjack/device/writer.26.rb	LOI	killed	4 (of 17)	0.041092s
lumberjack/device/writer.27.rb	LOI	survived	17 (of 17)	0.013922s
lumberjack/device/writer.28.rb	LOI	survived	17 (of 17)	0.044173s
lumberjack/device/writer.29.rb	LOI	survived	17 (of 17)	0.013687s
lumberjack/device/writer.30.rb	LOI	killed	1 (of 17)	0.05101s
lumberjack/device/writer.31.rb	LOI	killed	1 (of 17)	0.056578s
lumberjack/device/writer.32.rb	LOI	survived	17 (of 17)	0.013644s
lumberjack/device/writer.33.rb	LOI	survived	17 (of 17)	0.016998s
lumberjack/device/writer.34.rb	LOI	killed	5 (of 17)	0.060214s
lumberjack/device/writer.35.rb	LOI	killed	3 (of 17)	0.054294s
lumberjack/device/writer.36.rb	LOI	survived	17 (of 17)	0.013668s
lumberjack/device/writer.37.rb	SAOR	killed	2 (of 17)	0.047751s
lumberjack/device/writer.38.rb	SAOR	killed	2 (of 17)	0.062165s
lumberjack/device/writer.39.rb	SAOR	killed	2 (of 17)	0.050556s
lumberjack/device/writer.40.rb	SAOR	killed	2 (of 17)	0.053013s
lumberjack/device/writer.41.rb	SAOR	killed	2 (of 17)	0.055673s
lumberjack/device/writer.42.rb	SAOR	killed	2 (of 17)	0.051677s
lumberjack/device/writer.43.rb	SAOR	killed	2 (of 17)	0.066779s
lumberjack/device/writer.44.rb	SAOR	killed	2 (of 17)	0.049809s
lumberjack/device/writer.45.rb	SAOR	killed	2 (of 17)	0.055495s
lumberjack/device/writer.46.rb	SAOR	killed	2 (of 17)	0.222802s
lumberjack/device/writer.47.rb	SAOR	killed	2 (of 17)	0.062606s
lumberjack/device/writer.48.rb	SAOR	killed	2 (of 17)	0.050596s
lumberjack/device/writer.49.rb	UAOI	killed	4 (of 17)	0.053885s
lumberjack/device/writer.50.rb	UAOI	survived	17 (of 17)	0.013633s
lumberjack/device/writer.51.rb	UAOI	survived	17 (of 17)	0.015908s
lumberjack/device/writer.52.rb	UAOI	survived	17 (of 17)	0.011827s
lumberjack/device/writer.53.rb	UAOI	killed	1 (of 17)	0.048166s
lumberjack/device/writer.54.rb	UAOI	killed	1 (of 17)	0.061951s
lumberjack/device/writer.55.rb	UAOI	killed	3 (of 17)	0.049556s
lumberjack/device/writer.56.rb	UAOI	survived	17 (of 17)	0.034172s
lumberjack/formatter.0.rb	RER	stillborn	n/a	n/a
lumberjack/formatter.1.rb	RER	stillborn	n/a	n/a
lumberjack/formatter.2.rb	ROR	stillborn	n/a	n/a
lumberjack/formatter.3.rb	ROR	stillborn	n/a	n/a
lumberjack/formatter.4.rb	ROR	stillborn	n/a	n/a
lumberjack/formatter.5.rb	ROR	stillborn	n/a	n/a
lumberjack/formatter.6.rb	ROR	stillborn	n/a	n/a

lumberjack/formatter.7.rb	LOI	stillborn	n/a	n/a
lumberjack/formatter.8.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.9.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.10.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.11.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.12.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.13.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.14.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.15.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.16.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.17.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.18.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.19.rb	SAOR	stillborn	n/a	n/a
lumberjack/formatter.20.rb	UAOI	stillborn	n/a	n/a
lumberjack/formatter/pretty_print_formatter.0.rb	LOI	survived	6 (of 6)	0.007463s
lumberjack/formatter/pretty_print_formatter.1.rb	UAOI	survived	6 (of 6)	0.0064s
lumberjack/log_entry.0.rb	BAOR	killed	9 (of 14)	0.046028s
lumberjack/log_entry.1.rb	BAOR	killed	9 (of 14)	0.03789s
lumberjack/log_entry.2.rb	BAOR	killed	9 (of 14)	0.045772s
lumberjack/log_entry.3.rb	BAOR	killed	9 (of 14)	0.043216s
lumberjack/log_entry.4.rb	COI	survived	14 (of 14)	0.010175s
lumberjack/log_entry.5.rb	COI	killed	4 (of 14)	0.023405s
lumberjack/log_entry.6.rb	COI	killed	9 (of 14)	0.039228s
lumberjack/log_entry.7.rb	COI	killed	9 (of 14)	0.034186s
lumberjack/log_entry.8.rb	COI	killed	9 (of 14)	0.040976s
lumberjack/log_entry.9.rb	COI	killed	9 (of 14)	0.045885s
lumberjack/log_entry.10.rb	COI	killed	9 (of 14)	0.04208s
lumberjack/log_entry.11.rb	COI	killed	9 (of 14)	0.033595s
lumberjack/log_entry.12.rb	COI	killed	9 (of 14)	0.045112s
lumberjack/log_entry.13.rb	COI	killed	9 (of 14)	0.033509s
lumberjack/log_entry.14.rb	LOI	survived	14 (of 14)	0.008998s
lumberjack/log_entry.15.rb	LOI	killed	4 (of 14)	0.041213s
lumberjack/log_entry.16.rb	LOI	killed	9 (of 14)	0.037634s
lumberjack/log_entry.17.rb	LOI	killed	9 (of 14)	0.03329s
lumberjack/log_entry.18.rb	LOI	killed	9 (of 14)	0.033316s
lumberjack/log_entry.19.rb	LOI	killed	9 (of 14)	0.033246s
lumberjack/log_entry.20.rb	LOI	killed	9 (of 14)	0.043019s
lumberjack/log_entry.21.rb	LOI	killed	9 (of 14)	0.03432s
lumberjack/log_entry.22.rb	LOI	killed	9 (of 14)	0.035744s
lumberjack/log_entry.23.rb	LOI	killed	9 (of 14)	0.045988s
lumberjack/log_entry.24.rb	LOI	killed	9 (of 14)	0.038666s
lumberjack/log_entry.25.rb	UAOI	survived	14 (of 14)	0.008523s
lumberjack/log_entry.26.rb	UAOI	killed	4 (of 14)	0.042025s
lumberjack/log_entry.27.rb	UAOI	killed	9 (of 14)	0.037924s
lumberjack/log_entry.28.rb	UAOI	killed	9 (of 14)	0.034687s
lumberjack/log_entry.29.rb	UAOI	killed	9 (of 14)	0.032845s
lumberjack/log_entry.30.rb	UAOI	killed	9 (of 14)	0.036608s
lumberjack/log_entry.31.rb	UAOI	killed	9 (of 14)	0.040664s
lumberjack/log_entry.32.rb	UAOI	killed	9 (of 14)	0.033122s
lumberjack/log_entry.33.rb	UAOI	killed	9 (of 14)	0.036054s
lumberjack/log_entry.34.rb	UAOI	killed	9 (of 14)	0.04463s
lumberjack/log_entry.35.rb	UAOI	killed	9 (of 14)	0.038385s
lumberjack/log_entry.36.rb	UAOI	killed	9 (of 14)	0.042035s
lumberjack/logger.0.rb	BAOR	killed	30 (of 42)	0.369993s
lumberjack/logger.1.rb	BAOR	killed	30 (of 42)	0.378206s
lumberjack/logger.2.rb	BAOR	killed	30 (of 42)	0.373693s
lumberjack/logger.3.rb	BAOR	killed	30 (of 42)	0.383526s
lumberjack/logger.4.rb	BAOR	killed	19 (of 42)	0.219262s
lumberjack/logger.5.rb	BAOR	killed	19 (of 42)	0.235074s
lumberjack/logger.6.rb	BAOR	killed	19 (of 42)	0.220498s
lumberjack/logger.7.rb	BAOR	killed	19 (of 42)	0.239162s
lumberjack/logger.8.rb	COI	killed	30 (of 42)	0.370531s
lumberjack/logger.9.rb	COI	killed	30 (of 42)	0.392605s
lumberjack/logger.10.rb	COI	killed	30 (of 42)	0.388109s
lumberjack/logger.11.rb	COI	killed	30 (of 42)	0.377914s
lumberjack/logger.12.rb	COI	killed	30 (of 42)	0.374336s
lumberjack/logger.13.rb	COI	killed	30 (of 42)	0.37722s
lumberjack/logger.14.rb	COI	killed	13 (of 42)	0.072458s
lumberjack/logger.15.rb	COI	killed	13 (of 42)	0.070231s
lumberjack/logger.16.rb	COI	killed	30 (of 42)	0.373353s
lumberjack/logger.17.rb	COI	killed	26 (of 42)	0.377374s
lumberjack/logger.18.rb	COI	killed	13 (of 42)	0.059711s
lumberjack/logger.19.rb	COI	killed	30 (of 42)	0.375764s
lumberjack/logger.20.rb	COI	killed	30 (of 42)	0.391573s
lumberjack/logger.21.rb	COR	killed	6 (of 42)	0.058464s

lumberjack/logger.22.rb	COR	killed	6 (of 42)	0.068451s
lumberjack/logger.23.rb	COR	killed	9 (of 42)	0.068908s
lumberjack/logger.24.rb	COR	killed	9 (of 42)	0.073392s
lumberjack/logger.25.rb	COR	killed	6 (of 42)	0.061064s
lumberjack/logger.26.rb	COR	killed	6 (of 42)	0.075912s
lumberjack/logger.27.rb	COR	killed	30 (of 42)	0.375719s
lumberjack/logger.28.rb	COR	killed	23 (of 42)	0.374488s
lumberjack/logger.29.rb	COR	killed	13 (of 42)	0.061s
lumberjack/logger.30.rb	COR	killed	13 (of 42)	0.067473s
lumberjack/logger.31.rb	COR	killed	1 (of 42)	0.070465s
lumberjack/logger.32.rb	COR	killed	4 (of 42)	0.054195s
lumberjack/logger.33.rb	RER	killed	30 (of 42)	0.384763s
lumberjack/logger.34.rb	RER	killed	30 (of 42)	0.374885s
lumberjack/logger.35.rb	RER	killed	30 (of 42)	0.398448s
lumberjack/logger.36.rb	RER	killed	30 (of 42)	0.37642s
lumberjack/logger.37.rb	RER	killed	30 (of 42)	0.382471s
lumberjack/logger.38.rb	RER	killed	30 (of 42)	0.37356s
lumberjack/logger.39.rb	RER	killed	30 (of 42)	0.40064s
lumberjack/logger.40.rb	RER	killed	30 (of 42)	0.377017s
lumberjack/logger.41.rb	RER	killed	30 (of 42)	0.369194s
lumberjack/logger.42.rb	RER	killed	30 (of 42)	0.396413s
lumberjack/logger.43.rb	RER	killed	30 (of 42)	0.378195s
lumberjack/logger.44.rb	RER	killed	10 (of 42)	0.059824s
lumberjack/logger.45.rb	RER	killed	30 (of 42)	0.368754s
lumberjack/logger.46.rb	RER	killed	10 (of 42)	0.066534s
lumberjack/logger.47.rb	RER	killed	13 (of 42)	0.058283s
lumberjack/logger.48.rb	RER	killed	13 (of 42)	0.079731s
lumberjack/logger.49.rb	RER	killed	3 (of 42)	0.064056s
lumberjack/logger.50.rb	RER	killed	5 (of 42)	0.064657s
lumberjack/logger.51.rb	RER	killed	30 (of 42)	0.380784s
lumberjack/logger.52.rb	RER	killed	19 (of 42)	0.217525s
lumberjack/logger.53.rb	ROR	killed	30 (of 42)	0.398289s
lumberjack/logger.54.rb	ROR	killed	30 (of 42)	0.404972s
lumberjack/logger.55.rb	ROR	killed	30 (of 42)	0.38789s
lumberjack/logger.56.rb	ROR	killed	30 (of 42)	0.376413s
lumberjack/logger.57.rb	ROR	killed	30 (of 42)	0.401023s
lumberjack/logger.58.rb	ROR	killed	30 (of 42)	0.39135s
lumberjack/logger.59.rb	ROR	killed	30 (of 42)	0.386803s
lumberjack/logger.60.rb	ROR	killed	30 (of 42)	0.398402s
lumberjack/logger.61.rb	ROR	killed	30 (of 42)	0.374591s
lumberjack/logger.62.rb	ROR	killed	30 (of 42)	0.388513s
lumberjack/logger.63.rb	ROR	killed	30 (of 42)	0.373676s
lumberjack/logger.64.rb	ROR	killed	30 (of 42)	0.39853s
lumberjack/logger.65.rb	ROR	killed	30 (of 42)	0.371348s
lumberjack/logger.66.rb	ROR	killed	30 (of 42)	0.389827s
lumberjack/logger.67.rb	ROR	killed	30 (of 42)	0.368357s
lumberjack/logger.68.rb	ROR	killed	30 (of 42)	0.395689s
lumberjack/logger.69.rb	ROR	killed	30 (of 42)	0.366746s
lumberjack/logger.70.rb	ROR	killed	30 (of 42)	0.367666s
lumberjack/logger.71.rb	ROR	killed	30 (of 42)	0.397901s
lumberjack/logger.72.rb	ROR	killed	30 (of 42)	0.369006s
lumberjack/logger.73.rb	ROR	killed	30 (of 42)	0.385692s
lumberjack/logger.74.rb	ROR	killed	30 (of 42)	0.369903s
lumberjack/logger.75.rb	ROR	killed	30 (of 42)	0.396288s
lumberjack/logger.76.rb	ROR	killed	30 (of 42)	0.36847s
lumberjack/logger.77.rb	ROR	killed	30 (of 42)	0.384236s
lumberjack/logger.78.rb	ROR	killed	10 (of 42)	0.058373s
lumberjack/logger.79.rb	ROR	killed	10 (of 42)	0.066963s
lumberjack/logger.80.rb	ROR	killed	10 (of 42)	0.06593s
lumberjack/logger.81.rb	ROR	killed	30 (of 42)	0.374199s
lumberjack/logger.82.rb	ROR	killed	30 (of 42)	0.391083s
lumberjack/logger.83.rb	ROR	killed	10 (of 42)	0.044381s
lumberjack/logger.84.rb	ROR	killed	10 (of 42)	0.065929s
lumberjack/logger.85.rb	ROR	killed	10 (of 42)	0.054843s
lumberjack/logger.86.rb	ROR	killed	30 (of 42)	0.391785s
lumberjack/logger.87.rb	ROR	killed	30 (of 42)	0.367785s
lumberjack/logger.88.rb	ROR	killed	13 (of 42)	0.074034s
lumberjack/logger.89.rb	ROR	killed	13 (of 42)	0.061249s
lumberjack/logger.90.rb	ROR	killed	15 (of 42)	0.052919s
lumberjack/logger.91.rb	ROR	killed	13 (of 42)	0.074435s
lumberjack/logger.92.rb	ROR	killed	13 (of 42)	0.064542s
lumberjack/logger.93.rb	ROR	killed	3 (of 42)	0.074131s
lumberjack/logger.94.rb	ROR	killed	3 (of 42)	0.047667s
lumberjack/logger.95.rb	ROR	killed	3 (of 42)	0.072534s
lumberjack/logger.96.rb	ROR	killed	3 (of 42)	0.051884s

lumberjack/logger.97.rb	ROR	killed	3 (of 42)	0.063723s
lumberjack/logger.98.rb	ROR	killed	19 (of 42)	0.211426s
lumberjack/logger.99.rb	ROR	killed	19 (of 42)	0.231857s
lumberjack/logger.100.rb	ROR	killed	19 (of 42)	0.224898s
lumberjack/logger.101.rb	ROR	killed	30 (of 42)	0.376167s
lumberjack/logger.102.rb	ROR	killed	30 (of 42)	0.3812s
lumberjack/logger.103.rb	LOI	killed	17 (of 42)	0.066705s
lumberjack/logger.104.rb	LOI	killed	30 (of 42)	0.37896s
lumberjack/logger.105.rb	LOI	killed	30 (of 42)	0.377185s
lumberjack/logger.106.rb	LOI	killed	30 (of 42)	0.378877s
lumberjack/logger.107.rb	LOI	killed	30 (of 42)	0.386792s
lumberjack/logger.108.rb	LOI	killed	30 (of 42)	0.384307s
lumberjack/logger.109.rb	LOI	killed	13 (of 42)	0.061045s
lumberjack/logger.110.rb	LOI	killed	17 (of 42)	0.059316s
lumberjack/logger.111.rb	LOI	killed	30 (of 42)	0.383354s
lumberjack/logger.112.rb	LOI	killed	30 (of 42)	0.379619s
lumberjack/logger.113.rb	LOI	killed	26 (of 42)	0.374462s
lumberjack/logger.114.rb	LOI	killed	13 (of 42)	0.076201s
lumberjack/logger.115.rb	LOI	killed	30 (of 42)	0.385935s
lumberjack/logger.116.rb	LOI	killed	30 (of 42)	0.379952s
lumberjack/logger.117.rb	SAOR	killed	13 (of 42)	0.061966s
lumberjack/logger.118.rb	SAOR	killed	13 (of 42)	0.06411s
lumberjack/logger.119.rb	SAOR	killed	13 (of 42)	0.054533s
lumberjack/logger.120.rb	SAOR	killed	13 (of 42)	0.062032s
lumberjack/logger.121.rb	SAOR	killed	13 (of 42)	0.058688s
lumberjack/logger.122.rb	SAOR	killed	13 (of 42)	0.059963s
lumberjack/logger.123.rb	SAOR	killed	18 (of 42)	0.074583s
lumberjack/logger.124.rb	SAOR	killed	18 (of 42)	0.073438s
lumberjack/logger.125.rb	SAOR	killed	18 (of 42)	0.077787s
lumberjack/logger.126.rb	SAOR	killed	13 (of 42)	0.059742s
lumberjack/logger.127.rb	SAOR	killed	13 (of 42)	0.057999s
lumberjack/logger.128.rb	SAOR	killed	18 (of 42)	0.073847s
lumberjack/logger.129.rb	UAOI	killed	17 (of 42)	0.066497s
lumberjack/logger.130.rb	UAOI	killed	30 (of 42)	0.380838s
lumberjack/logger.131.rb	UAOI	killed	30 (of 42)	0.374391s
lumberjack/logger.132.rb	UAOI	killed	30 (of 42)	0.397178s
lumberjack/logger.133.rb	UAOI	killed	30 (of 42)	0.375828s
lumberjack/logger.134.rb	UAOI	killed	30 (of 42)	0.37668s
lumberjack/logger.135.rb	UAOI	killed	13 (of 42)	0.058884s
lumberjack/logger.136.rb	UAOI	killed	17 (of 42)	0.068345s
lumberjack/logger.137.rb	UAOI	killed	26 (of 42)	0.371022s
lumberjack/logger.138.rb	UAOI	killed	13 (of 42)	0.081864s
lumberjack/logger.139.rb	UAOI	killed	13 (of 42)	0.061045s
lumberjack/logger.140.rb	UAOI	killed	30 (of 42)	0.380246s
lumberjack/template.0.rb	BAOR	killed	6 (of 10)	0.026612s
lumberjack/template.1.rb	BAOR	killed	6 (of 10)	0.029727s
lumberjack/template.2.rb	BAOR	killed	6 (of 10)	0.02714s
lumberjack/template.3.rb	BAOR	killed	6 (of 10)	0.026848s
lumberjack/template.4.rb	BAOR	killed	6 (of 10)	0.025926s
lumberjack/template.5.rb	BAOR	killed	6 (of 10)	0.028208s
lumberjack/template.6.rb	BAOR	killed	6 (of 10)	0.026237s
lumberjack/template.7.rb	BAOR	killed	6 (of 10)	0.032601s
lumberjack/template.8.rb	BAOR	killed	6 (of 10)	0.027332s
lumberjack/template.9.rb	BAOR	killed	6 (of 10)	0.027379s
lumberjack/template.10.rb	BAOR	killed	6 (of 10)	0.034428s
lumberjack/template.11.rb	BAOR	killed	6 (of 10)	0.026763s
lumberjack/template.12.rb	BAOR	killed	7 (of 10)	0.028974s
lumberjack/template.13.rb	BAOR	killed	7 (of 10)	0.032258s
lumberjack/template.14.rb	BAOR	killed	7 (of 10)	0.055645s
lumberjack/template.15.rb	BAOR	killed	7 (of 10)	0.028531s
lumberjack/template.16.rb	BAOR	killed	6 (of 10)	0.025225s
lumberjack/template.17.rb	BAOR	killed	6 (of 10)	0.030136s
lumberjack/template.18.rb	BAOR	killed	6 (of 10)	0.023759s
lumberjack/template.19.rb	BAOR	killed	6 (of 10)	0.028458s
lumberjack/template.20.rb	BAOR	killed	6 (of 10)	0.037683s
lumberjack/template.21.rb	BAOR	killed	6 (of 10)	0.032558s
lumberjack/template.22.rb	BAOR	killed	6 (of 10)	0.036031s
lumberjack/template.23.rb	BAOR	killed	6 (of 10)	0.038478s
lumberjack/template.24.rb	COR	killed	6 (of 10)	0.029864s
lumberjack/template.25.rb	COR	killed	6 (of 10)	0.024873s
lumberjack/template.26.rb	COR	survived	10 (of 10)	0.007697s
lumberjack/template.27.rb	COR	killed	6 (of 10)	0.032504s
lumberjack/template.28.rb	COR	killed	6 (of 10)	0.03798s
lumberjack/template.29.rb	COR	killed	6 (of 10)	0.030004s
lumberjack/template.30.rb	RER	killed	7 (of 10)	0.034531s

lumberjack/template.31.rb	RER	killed	6 (of 10)	0.039819s
lumberjack/template.32.rb	ROR	killed	6 (of 10)	0.035005s
lumberjack/template.33.rb	ROR	killed	7 (of 10)	0.036146s
lumberjack/template.34.rb	ROR	killed	6 (of 10)	0.036642s
lumberjack/template.35.rb	ROR	survived	10 (of 10)	0.009347s
lumberjack/template.36.rb	ROR	killed	6 (of 10)	0.030394s
lumberjack/template.37.rb	LOI	killed	6 (of 10)	0.027855s
lumberjack/template.38.rb	UAOI	killed	6 (of 10)	0.025358s
lumberjack/template.39.rb	UAOI	killed	6 (of 10)	0.038565s

Appendix 6: Links to online MultiMutiny reports

URLs for uploaded HTML reports and live reports on the MultiMutiny cloud system for the codebases mentioned in the Evaluation section are given below.

As with Appendix 5, the first project's report was generated without using CI features, while the other two show a passing and a failing CI build in that order.

event_bus

HTML	https://waterson.co/project/reports/event_bus.html
Cloud	https://multimutiny-cloud-reports.appspot.com/report/ahtlfm11bHRpbXV0aW55LWNsb3VkLXJlcG9ydHNyEwsSBlJlcG9ydBiAgICA7bGDCgw/

full_name

HTML	https://waterson.co/project/reports/full_name.html
Cloud	https://multimutiny-cloud-reports.appspot.com/report/ahtlfm11bHRpbXV0aW55LWNsb3VkLXJlcG9ydHNyEwsSBlJlcG9ydBiAgICAq_OHCgw/

lumberjack

HTML	https://waterson.co/project/reports/lumberjack.html
Cloud	https://multimutiny-cloud-reports.appspot.com/report/ahtlfm11bHRpbXV0aW55LWNsb3VkLXJlcG9ydHNyEwsSBlJlcG9ydBiAgICAr8iACgw/

Bibliography

[1] B. Hetzel, *The Complete Guide to Software Testing*, Second. QED Information Sciences, Inc., 1988.

[2] Smithsonian National Museum of American History, "Log Book With Computer Bug." [Online]. Available: http://americanhistory.si.edu/collections/search/object/nmah_334663. [Accessed: 07-Feb-2016].

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Pearson Education Ltd, 2003.

[5] Centre for the Protection of National Infrastructure, "Human factors in CCTV control rooms: A best practice guide," 2014. [Online]. Available: https://www.cpni.gov.uk/documents/publications/2014/2014001-human_factors_cctv_control_rooms.pdf. [Accessed: Mar-2016].

[6] Python Software Foundation, "Python 3 Documentation: 26.4. unittest — Unit testing framework." [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed: 05-Apr-2016].

[7] V. Massol and T. Husted, *JUnit in Action*. Manning Publications Co., 2004.

[8] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North, *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Bookshelf, 2010.

[9] J. Tromp and G. Farnebäck, "Combinatorics of Go," *Computers and Games*. pp. 84–99, May-2006.

[10] Q. Yang, J. J. Li, and D. M. Weiss, "A Survey of Coverage-Based Testing Tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[11] B. Marick, "How to Misuse Code Coverage," *Proceedings of the 16th International Conference on Testing Computer Software*. pp. 16–18, Jun-1999.

[12] J. Offutt, "A Mutation Carol: Past, Present and Future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[13] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, Jun. 2010.

[14] J. Offutt, R. Pargas, S. Fichter, and P. Khambekar, "Mutation Testing of Software Using a MIMD Computer," *Proceedings of the International Conference on Parallel Processing*. pp. II 257–266, 1992.

[15] J. V. Carreira, D. Costa, and J. G. Silva, "Fault injection spot-checks computer system dependability," *IEEE Spectrum*. pp. 50–55, Aug-1999.

[16] A. Derezinska and K. Halas, "Operators for Mutation Testing of Python Programs," *Warsaw University of Technology, ICS Research Report*. 2014.

[17] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5. pp. 649–678, Sep-2011.

[18] J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*. pp. 45–55, Oct-2000.

[19] Kennedy Space Center, "Mariner 1 Launch." [Online]. Available: http://grin.hq.nasa.gov/ABSTRACTS/GPN-2000-000608.html. [Accessed: 07-Feb-2016].

[20] NASA Space Science Data Center, "Mariner 1." [Online]. Available: http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1. [Accessed: 04-Feb-2016].

[21] L. Inozemtseva and R. Holmes, "Coverage Is Not Strongly Correlated with Test Suite Effectiveness," *Proceedings of the 36th International Conference on Software Engineering*. pp. 435–445, 2014.

[22] J. H. Andrews, L. C. Briand, and true Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proceedings of the 27th International Conference on Software Engineering*. 2005.

[23] A. Derezinska and K. Halas, "Analysis of Mutation Operators for the Python Language," *Proceedings of the Ninth International Conference on Dependability and Complex Systems*. 2014.

[24] D. Stahl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *The Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[25] S. Stolberg, "Enabling Agile Testing Through Continuous Integration," *Agile Conference*. 2009.

[26] M. Fowler, "Continuous Integration," 01-May-2006. [Online]. Available: http://www.martinfowler.com/

articles/continuousIntegration.html. [Accessed: 06-Apr-2016].

[27] "CruiseControl." [Online]. Available: http://cruisecontrol.sourceforge.net/. [Accessed: 06-Apr-2016].

[28] "Jenkins: Build great things at any scale." [Online]. Available: https://jenkins.io/. [Accessed: 06-Apr-2016].

[29] Atlassian Pty Ltd, "Bamboo: Build, Test, Deploy." [Online]. Available: https://www.atlassian.com/software/bamboo. [Accessed: 06-Apr-2016].

[30] Travis CI GmbH, "Travis CI: Test and Deploy with Confidence." [Online]. Available: https://travis-ci.org/. [Accessed: 06-Apr-2016].

[31] Pragmatic Automation, "Bubble, Bubble, Build's In Trouble." [Online]. Available: https://web.archive.org/web/20140104051209/http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc. [Accessed: 07-Feb-2016].

[32] R. O. Rogers, "Scaling Continuous Integration," *Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 5th International Conference*, Jun. 2004.

[33] J. Shore and S. Warden, *The Art of Agile Development*. O'Reilly, 2007.

[34] J. Offutt and W. M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[35] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, Oct. 2001.

[36] Lemur Heavy Industries, "Coveralls: Deliver better code." [Online]. Available: https://coveralls.io/. [Accessed: 06-Apr-2016].

[37] Travis CI GmbH, "Customizing the Build — Travis CI." [Online]. Available: https://docs.travis-ci.com/user/customizing-the-build/. [Accessed: 10-Apr-2016].

[38] Travis CI GmbH, "The Build Environment — Travis CI." [Online]. Available: https://docs.travis-ci.com/user/ci-environment/. [Accessed: 10-Apr-2016].

[39] M. Schirp, "Mutant: Mutation testing for Ruby." [Online]. Available: https://github.com/mbj/mutant. [Accessed: 10-Apr-2016].

[40] L. Rose, "Mutiny: An experimental mutation testing framework for Ruby." [Online]. Available: http://www.mutiny.eu/. [Accessed: 10-Apr-2016].

[41] H. Kniberg, "Kanban vs Scrum: How to make the most of both." Jun-2009.

[42] M. Schirp, "Mutant README." [Online]. Available: https://github.com/mbj/mutant/blob/b8c8f65/README.md. [Accessed: 16-Apr-2016].

[43] Google Inc., "Google Cloud Platform: App Engine Quotas." [Online]. Available: https://cloud.google.com/appengine/docs/quotas. [Accessed: 16-Apr-2016].

[44] Google Inc., "Google Cloud Platform: The Python DB Client Library for Cloud Datastore." [Online]. Available: https://cloud.google.com/appengine/docs/python/datastore/. [Accessed: 16-Apr-2016].

[45] Codecov LLC, "Codecov: Continuous code coverage." [Online]. Available: https://codecov.io/. [Accessed: 23-Apr-2016].

[46] G. Finn and E. Horowitz, "A linear time approximation algorithm for multiprocessor scheduling," *BIT Numerical Mathematics*, vol. 19, no. 3, pp. 312–320, Sep. 1979.

[47] Microsoft Developer Network, "ASCII Character Codes Chart 1." [Online]. Available: https://msdn.microsoft.com/en-us/library/60ecse8t.aspx. [Accessed: 03-Feb-2016].