DELFT UNIVERSITY OF TECHNOLOGY

BACHELOR THESIS PROJECT

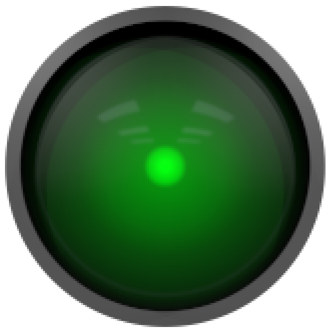
AuTA

Authors:
Luc EVERSE
Tim VAN DER HORST
Erik OUDSEN
Ewoud RUIGHAVER

Client:
Ir. Otto VISSER

Coach:
Dr. Annibale PANICHELLA

Coordinator:
Dr. Huijuan WANG



June 27, 2019

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science

Software Technology

AuTA

by
Luc EVERSE
Tim VAN DER HORST
Erik OUDSEN
Ewoud RUIGHAVER

Due to the dramatic increase in enrollments in the TU Delft Bachelor of Computer Science, the workload for teaching assistants and instructors has skyrocketed. To reduce this workload, automated tools can be used to make the grading process easier. This paper describes the development of AuTA (Automatic Teaching Assistant), a tool that will help instructors and teaching assistants analyze and grade programming assignments and provide useful feedback to the student.

Contents

Abstract **i**

1 Introduction **7**

2 Research Report **8**
- 2.1 Problem definition . . . 8
- 2.2 Project requirements . . . 9
- 2.3 State of the art . . . 11
- 2.4 Design choices . . . 16
- 2.5 Future of Computer Science education . . . 21
- 2.6 Conclusion . . . 21

3 Architecture **22**
- 3.1 Modular design . . . 22
- 3.2 Kubernetes . . . 23
- 3.3 REST API . . . 23
- 3.4 AuTA.js . . . 23
- 3.5 Core Plugins . . . 24
- 3.6 SAML Single-Sign On . . . 24
- 3.7 Trampoline . . . 25
- 3.8 JUMP . . . 26
- 3.9 Worker - Core communication . . . 27
- 3.10 Entity structure . . . 27
- 3.11 Metric calculation . . . 28
- 3.12 Analyzers . . . 29
- 3.13 Metric scripts . . . 30
- 3.14 Report . . . 31
- 3.15 Feedback generation . . . 32
- 3.16 Method names . . . 32
- 3.17 Language detection . . . 34
- 3.18 Suppressions . . . 34
- 3.19 Security . . . 34

4 Tools **37**
- 4.1 Test Smells Detector . . . 37
- 4.2 Lizard . . . 37
- 4.3 QDox . . . 37
- 4.4 PlantUML . . . 38
- 4.5 CK . . . 38
- 4.6 Radon . . . 38
- 4.7 PyLint . . . 38
- 4.8 Profanity Checker . . . 38

5 Performance **40**
5.1 Method 40
5.2 Discussion 41

6 Process **42**
6.1 Research phase 42
6.2 Production phase and challenges 42
6.3 Evaluation 47

7 Ethics **48**
7.1 Human vs. machine 48
7.2 Personal information and data retention 48

8 Discussion **49**
8.1 Requirements 49
8.2 Security 50
8.3 Feedback quality 50
8.4 Generating grades 50

9 Future work and recommendations **52**
9.1 System score generation 52
9.2 More languages 52
9.3 Easy Docker 52
9.4 Annotation suppressions 53
9.5 Entity tree 53
9.6 Machine Learning 53
9.7 Support for huge projects 54
9.8 Multiple Languages 54
9.9 Grading 55
9.10 Future of AuTA as part of Labrador 55

10 Conclusion **56**

Acknowledgements **57**

Appendices **58**

A MoSCoW **59**

B Data management plan **62**

C Architectural diagrams **67**

D Labrador overview **71**

E AuTA Submission flows **73**

F Metric overview **75**

G Manual **78**
G.1 Cluster setup 78
G.2 System setup 79
G.3 Devops 81

G.4 External Service Integration 81

H Tool metrics **83**
H.1 Test Smells Metrics 83
H.2 CK Metrics 84

I Project statistics **86**

J Example Reports **87**

K UI screenshots **90**

L Repositories **99**
L.1 Links 99

M Research phase **100**
M.1 Interviews 100

N Stress Test Results **104**

O SIG feedback **106**
O.1 First feedback 106

P Infosheet **108**

Bibliography **110**

List of Figures

2.1 Diagram of new worker structure (non standard UML) 19

3.1 The new AuTA structure. (Uses PlantUML syntax [33]) 22
3.2 AuTA SAML 2.0 SSO authentication flow. (Uses PlantUML syntax [33]) 25
3.3 A JUMP message 26
3.4 Sequence diagram showing key exchange flow between workers and core. (Uses PlantUML syntax [33]) 27
3.5 Entity structure UML. (Uses PlantUML syntax [33]) 28
3.6 Sequence diagram for submission message processing. (Uses PlantUML syntax [33]) 29
3.7 Sequence diagram example for analyzer (Lizard). (Uses PlantUML syntax [33]) 30
3.8 Sequence diagram for the core. (Uses PlantUML syntax [33]) 32

C.1 System architecture as of AuTA 1.0. (Uses PlantUML syntax [33]) . . . 67
C.2 System architecture in AuTA 2.0. (Uses PlantUML syntax [33]) 68
C.3 Activity diagram for Trampoline. (Uses PlantUML syntax [33]) 69
C.4 Class diagram for analyzers and checkers in the worker. (Uses PlantUML syntax [33]) 70
C.5 Activity diagram for the process of merging method names. (Uses PlantUML syntax [33]) 70

D.1 Labrador simplified. (Not UML, just proposed overview drawings) . . 71
D.2 Labrador detailed. (Not UML, just proposed overview drawings) . . . 72

E.1 The flow of a submission through CPM. (Not UML, just proposed overview drawings) 73
E.2 The flow of a submission through GitLab. (Not UML, just proposed overview drawings) 74

J.1 An example of a report that is sent to the student directly (e.g. via GitLab) 87
J.2 An example of a report that is uploaded to CPM 88
J.3 An example histogram generated for cyclomatic complexity. Clicking on a histogram bar lists the entities with that cyclomatic complexity . . 89

K.1 The page that shows a list of all courses that have been created 90
K.2 The page for a specific course, where TAs or instructors can add assignments to that course, or add users to the course 90
K.3 The page where all assignments are listed, with the option to edit or clone an assignment (which creates a new assignment with the exact same metrics and language) 91

K.4 The assignment creation page, where the name of the assignment and language can be selected . . . 91
K.5 The page where metrics can be added to an assignment. The drop-down contains all metrics that are available for the assignment's language . . . 92
K.6 An example of a passing script that can be customized for each assignment . . . 92
K.7 The page that lists all submissions for a specific assignment. The image on the right side of each submission indicates the verdict associated with that submission - red is fail, blue is error, green is pass and yellow is pass with warnings . . . 93
K.8 The page for a specific submission. Each entity with warnings or failures is listed. Users can select whether they want to show warnings, tips or failures (or all of them). The status bar on the right shows information about how long the analysis took for that assignment . . . 94
K.9 The page for a specific submission, with tips showing . . . 95
K.10 The page where the user can generate histograms for each metric, where the bar height is equal to the total entity weight with that metric value. Entities are aggregated into bins with the same metric values . . 96
K.11 The page that shows all users that have been registered in the database. On this page, users can edit their passwords. Admins can create new users, or edit their roles. . . . 96
K.12 The page that lists all workers that are currently connected to the system 97
K.13 The general settings page . . . 97
K.14 The workers settings page . . . 97
K.15 The security settings page . . . 98
K.16 The integrations settings page . . . 98
K.17 The MongoDB settings page . . . 98

M.1 Calculating maintainability score figures, from [19] . . . 100

N.1 Graph of total time with varying number of workers and submissions 104
N.2 Graph of effective lines of code per second to analyze 200 simultaneous submissions with varying number of workers . . . 105
N.3 Boxplot of effective LOC/s with varying number of workers . . . 105

List of Tables

2.1 Source code properties and how they are measured, from [19] 15

3.1 Security agent concerns 36

5.1 Profiling comparison for small and large submissions 41

A.1 Must-have requirements 60
A.2 Should-have requirements 61
A.3 Could-have requirements 61
A.4 Won't-have requirements 61

G.1 Arguments for the worker 80
G.2 Arguments for the worker when starting from docker 80

I.1 Lines of code and test coverage for each project repository 86

Chapter 1

Introduction

Over the past few years, the number of students who have enrolled in the Computer Science program at the Delft University of Technology has increased significantly. Programming assignments are an important part of the Bachelor's degree and are used to help students practice the theory they learn during lectures. Many courses employ teaching assistants to grade or approve these assignments. The workload of both the teaching assistants and instructors has been increasing with the growing number of students, which can be detrimental to the quality of the education.

In order to maintain the quality of the study program and to reduce the workload of TAs and instructors, a number of tools have been developed at the TU Delft's Educational Innovation Projects Group. These tools help to automate the process of reviewing student submissions.

Our task was to develop a tool that could automatically check submissions of students and provide them with useful feedback or even a grade. The tool needed to be highly scalable and reliable in order to cope with a large number of students. This tool, which is called AuTA (Automated Teaching Assistant) was developed over the course of 10 weeks.

The first two weeks were dedicated to literature review – specifically how to generate useful feedback and other existing similar tools. This is elaborated on in Chapter 2. The following 8 weeks were dedicated to development. We explain our design in Chapter 3. Then we describe the tools that we have used to aid in reviewing the submitted code and their licenses in Chapter 4. We assess the performance of the product in Chapter 5. We also discuss our process during the research phase and during the production phase in Chapter 6. We discuss some ethical implications in Chapter 7. Next, in Chapter 8 we discuss some problems that we encountered during the project and give recommendations as to how to proceed with the tool in Chapter 9. We finalize the report by giving our conclusion on the project in Chapter 10.

Chapter 2

Research Report

In this chapter, we present the results of the literature review that preceded the development process. This chapter was written before we started the development process. The rest of the report was written after the development process.

2.1 Problem definition

In this section, we give a definition and context of the problem that needs to be solved.

2.1.1 Context

In 2018, the number of first-year students at the Computer Science and Engineering Bachelors nearly doubled compared to the year before (885 vs 450 students) [1]. Furthermore, the Delft University of Technology also wants to integrate programming skills in the curriculum for Bachelor programs at other faculties. This puts an extra strain on the limited resources that are available for checking and grading assignments. Currently, this is being solved by hiring more teaching assistants for each course to handle the increasing workload. However, this decreases the quality of the course as it will be harder to have every TA check the exercises in the same way. Students might, therefore, be tempted to continuously submit their assignments to different TAs until someone approves their code.

2.1.2 Definition

Students should get automated feedback for both smaller homework assignments and larger projects to reduce the workload on the teaching assistants and instructors. Many students do not have previous programming experience, which is why the system should return helpful feedback instead of raw numbers or style requirements without context.

In addition, each course has different requirements. The tool must, therefore, be easy to configure by teaching assistants and instructors. It should also integrate seamlessly with the various platforms the courses use to deliver their submissions such as Gitlab, WebLab, and CPM.

2.2 Project requirements

In this section, we discuss the process of gathering requirements for our project. To create the list of requirements we interviewed stakeholders to gain a better understanding of what is expected of the tool.

2.2.1 Stakeholder Interviews

This section contains summarized interviews. Longer versions can be found in appendix section M.1

Dr. Santosh Ilamparuthi - EEMCS Data Steward

We approached the EEMCS faculty data steward with a few questions regarding the usage and management of student data. The programming assignments students deliver may contain personally identifiable information such as student number, email address or name. Care has to be taken to maintain the integrity of the data, and ensure the data is protected against unauthorized change or access. In general, security has to be at the same level as other services that run behind the TU Delft firewall. Finally, all third-party tools need to be audited for any misuse of identifiable information.

Thomas Overklift - Head TA

Thomas is a Head TA for a number of courses in the Computer Science Bachelor Program. He indicated that feedback should be more detailed for smaller projects and should give useful tips to improve the students' style and general programming skills. There should be a trade-off between the amount of time needed to configure the tool to give more detailed feedback and the quality of feedback given.

For larger projects, the tool could be used to flag certain areas of code that could be problematic for both the TAs and students.

Frank Mulder - Instructor

Frank Mulder is an instructor for Object-Oriented Programming (OOP) and Introduction to Python Programming (IPP). For his introductory courses, feedback needs to be more human-readable and could be adjusted to give programming advice to students to improve their quality and skills. WebLab integration is very important for this instructor.

Maurício Aniche - Instructor

Maurício Aniche is a lecturer for the Software Quality and Testing course that is given in the fourth quarter of the first year. He wanted metrics that were specifically for test code, such as the STREW-J metric suite [2]. Secondly, he would like to give students feedback per metric on how they performed relative to other students because this might motivate them to improve. An overview of these statistics should be visible by the lecturer, to identify groups or students that are struggling. Thirdly, students should be able to run their code through the tool whenever they want.

Daniel Pelsmaeker / Danny Groenewegen / Elmer van Chastelet - WebLab

WebLab is a learning management system used for a number of courses at the TU Delft. Students are able to complete programming assignments in WebLab. Unit tests are then used to calculate a score for student assignments.

The developers of WebLab requested that we send them our API documentation so that they could add something to WebLab that would allow submissions to be sent to our tool. They also requested that we would generate course specific API tokens so that other courses can't be accessed if the right token is not present. Lastly, they requested that we return the results of our tool in JSON format instead of our HTML generated report.

Annibale Panichella - Project Coach

We had weekly meetings with Annibale Panichella, who may also use our tool next year. First, he mentioned that we should test the security of our system. He also mentioned that returning simple messages as feedback is often better than returning the value of the metric.

Otto Visser - Client

In addition to all other requirements, our client would like to be able to grade assignments based on their submission and a fixed rubric.

2.2.2 MoSCoW

In Appendix A we have listed the requirements as we think they should be prioritized, divided into categories. In this section, we will justify the choices made in classifying these requirements.

Must have

For our *must have*s, we have chosen requirements that are vital to AuTA.

Several categories stand out since most or all of their corresponding requirements are must-haves. The GDPR category is one of these. This contains requirements related to GDPR compliance and to be able to deploy AuTA as part of a course, it has to comply with these regulations.

The worker management category is similar. In AuTA 1.0, worker management was still unfinished with limited support for automatically starting and stopping workers depending on the current load. To be able to handle the large number of submissions that are expected when deployed as part of a course, current functionality needs to be expanded.

Should have

The requirements labeled as *should-have* are requirements with high priority, but are not necessarily vital to AuTA.

This, for instance, includes the requirement that there should be integration with the ED5 context group, who are researching a GitLab front-end. Since our front-end has the same capabilities as this front-end would have, it is not necessarily vital. However, since it would allow users to easily manage assignments directly from GitLab, this requirement is high-priority regardless.

Could have

Our *could have*s are requirements that could be implemented as they would be a nice addition to the product. However, they are not vital to the success of the project and will therefore most likely not be implemented within the given time frame.

Won't have

Our *won't have*s contain some items that were previous higher-priority requirements of AuTA 1.0. These were however dropped during the development of AuTA 2.0. We found that Scala was not used frequently enough to justify spending time on implementing support for it, for example.

2.3 State of the art

Many tools providing code quality assessment for software projects already exist. In this chapter, we explore the various solutions that are readily available and analyze their strengths and weaknesses. Due to the sheer volume of these tools, not all tools have been discussed: only those with a large enough user-base or those with features worth mentioning.

2.3.1 Static analysis tools

Existing static analysis tools could potentially be used by the project as an analysis back-end. Some tools were already used in the previous iteration of the software; these are named in Chapter 2.4.

Metrics analyzers

Many different metrics exist to measure the quality of code. Metrics can be grouped in metric suites, such as the Chidamber and Kemerer suite [3] or MOOD suite [4] for OOP languages. Many tools calculating different metrics already exist, such as JaSoMe[5] for Java, Lizard [6] for many different languages, and Radon [7] for Python. Most tools output results in an XML format, which is easy to parse. Due to the similar output format and mathematical nature of the metrics, tools are easy to interchange.

Linters and bug detectors

Linters and bug detectors help to improve code quality by searching for style violations and common programming mistakes. These tools mainly improve the maintainability of large software projects, with popular examples in the Java ecosystem being PMD [8], SpotBugs [9], and Checkstyle [10]. Their applicability to smaller source sets, such as homework submissions, may be limited.

2.3.2 Code analysis services

Some solutions exist that provide a centralized service that projects can be submitted to for review. These services run outside of the project's code base.

CS education

We found a literature survey by Keuning et al. [11] that reviewed 101 tools that could provide feedback to students. They stated that there were two types of feedback that could be returned to students: formative and summative feedback. Summative feedback is feedback that only returns a number towards the student. Although it might give an indication to the student how they are doing, Keuning et al. state that this is mostly superficial. Formative feedback is, as described by Shute et al. [12] "information communicated to the learner with the intention to modify his or her thinking or behavior for the purpose of improving learning". According to Keuning et al. formative feedback has a positive effect on the way feedback is used by students.

In the paper Keuning et al. describe that most tools focus on finding mistakes and not on providing the next step that might help the student. Thus they are mostly summative and therefore do not provide the most useful feedback. The tools that do provide formative feedback are often so focused on creating feedback for a specific assignment that they are not easily adaptable for instructors to apply on their own assignments.

JACK One tool that provides only summative feedback is JACK by Goedicke et al. [13]. This tool was used in the academic year of 2007 by the University of Duisburg-Essen. It has a lot of similarities to the current version of AuTA. It has the same structure for example: it has a web front-end through which code can be submitted towards a core component that contains a database. The core component forwards it to a backend checker (*worker* in AuTA) and gives back the results to the core component which returns the results towards the student. The creators of the tool described the testing of the tool as a success as the instructors were spending less time checking submitted answers, however, they acknowledged that was partly due to that students were sharing answers on online forums as to what solutions the system would accept and then submitting slightly altered versions to avoid plagiarism detection.

AutoTeach An example of a formative tool that Keuning et al. gave was AutoTeach [14] by Antonucci et al. This tool was developed for MOOCs (Massive Open Online Courses) as there is not always someone around that the students could ask questions to. The way they implemented the feedback was relatively simple: they allowed the instructor to put various 'hints' into the system. Then they gave the students a button which, when pressed, gave part of the solution to the assignment with the corresponding hint of the instructor.

We believe this will not be the best solution for AuTA as this opens the possibility for abuse: being presented with a button which essentially gives them the answer might discourage students from trying to make the assignment themselves (which is confirmed by Antonucci et al.) and will thus no longer learn anything from it.

EASy EASy stands for E-Assessment System which is a web-based online learning system to help learning programmers [15]. In this paper, they state that hardly any tools exist that can provide feedback that requires higher cognitive skills, and the ones that do exist do not come out of their experimental state. To provide feedback their tool makes use of a couple of other tools, mainly Checkstyle and FindBugs. They also use predefined JUnit test cases. These tools are however not as useful for beginners in programming as they might not be able to understand the results of these tools. Therefore, a transformation might be applied to the output of these tools to create more valuable feedback for the student.

FIT Another system that tries to get more formative feedback is FIT [16]. In their paper they state that creating detailed feedback is very time consuming, it might cost an instructor 100-1000 hours of work for one hour of instruction with detailed feedback [17]. This is because a system that can deliver such detailed feedback requires exact formalization and underlying knowledge of the assignment at hand. FIT tries to circumvent this by creating an AI that gives feedback. When a student submits an assignment, the AI compares it to the given template solutions and returns the most comparable solution. They hope that this will give the student insight into what they could have done better in their assignment. We believe this is not the case, as many students will simply copy the solution.

ELP ELP (Environment for Learning to Program) is a system that is also build to generate feedback for beginning students [18]. This system is a "fill in the gap" system: students are provided a template for which they have to fill in certain methods. It provides a number of feedback points such as unused variables and complex code. Most of these metrics can however also be retrieved from a simple Checkstyle check. Besides these metrics, they also use predefined tests to check whether the student has correctly implemented the assignment.

Evaluation In the paper by Keuning et al. they mention three types of Intelligent Tutoring Systems (ITSs): Model Tracing (MT), Constraint-based modeling (CBM), and Data Analysis (DA). We think the latter two will be useful for our tool. Model tracing is a technique that generates feedback based on the steps that the student took to come to a certain answer. This technique is not an option for our tool as we will only receive the final version of the product that the student hands in and not the intermediate steps they took.

The Constraint-based modeling technique is just checking the solution against certain constraints, such as if a function is not too complex. This feature is already available in AuTA 1.0. This feedback is, however, summative and therefore needs to be improved.

Data Analysis involves providing feedback by looking at solutions of previous students and generating hints from those as to how the code could be improved. Therefore we think that data analysis is also a technique that could be used in our tool. As many first-year courses have a large number of submissions, we could create a benchmark. This benchmark could be used to provide hints to students as to where they could still improve their code. It could, for example, give feedback saying that the amount of methods used is more than 95% of the solutions submitted by other students, it doesn't necessarily have to be wrong, but a TA knows that they can take a look at it.

Software Improvement Group

The Software Improvement Group (SIG) have published their methods for assessing the quality of software [19][20][21][22]. These methods are also implemented in their online tool Better Code Hub [23]. SIG assigns a maintainability score and compares their results with comparable projects. Their software only works with GitHub, unfortunately, and requires human attention for useful feedback.

Benchmarking method Their benchmarking method, from the paper by Alves et. al. [21] is summarized as follows:

1. For each system entity (e.g. methods, classes, files) a metric value and weight value is calculated. The weight value is normally the number of lines of code for that entity;
2. For each entity, calculate a relative weight $\frac{\text{Entity Weight}}{\text{Total System Weight}}$;
3. Aggregate weights of all entities per metric value (for example, entities with a cyclomatic complexity of 10 represent 22% of the total system code);
4. Repeat this for multiple systems;
5. Calculate thresholds that can be used to assign risk scores to systems by comparing different system's results.

No code actually needs to be stored for this method.

Maintainability score calculation Heitlager et. al. (also from SIG) present a method to measure the maintainability of software, using a similar method as described above [19].

1. Metric values are assigned to risk categories (low – high) according to certain threshold levels;
2. Relative weight of all code in each risk category is calculated using thresholds (see Figure M.1a in the Appendix);
3. A schema is used to calculate a source code property rating for the entire system (see Figure M.1b);
4. This is repeated for 5 different source code properties (see Table 2.1);
5. A total system score is calculated using a mapping of source code properties to maintainability sub-characteristics (see Figure M.1c).

Source code property	**Metrics used**
Volume	LOC, man-years
Complexity per unit	Cyclomatic Complexity
Duplication	Duplicated blocks >6 lines
Unit testing	Coverage, Number of assertions
Unit size	LOC per method/class

TABLE 2.1: Source code properties and how they are measured, from [19]

Evaluation Maintainability scores are not very useful for smaller projects, such as the programming assignments for first-year courses like Object Oriented Programming and Introduction to Python Programming. Solutions to these assignments generally only consist of a few small files, which makes maintainability less relevant.

However, maintainability is more important during larger projects such as the Bachelor Thesis Project, Context Project, and the Software Engineering Methods project. This could be a useful feature to implement in our tool and it is not very difficult to do.

Comparing a submission to a benchmark of previous assignments might be useful to identify possible problematic areas of code. For example, if a student's submission has a method which is more complex than 95% of all other submissions, the student could be notified that there are other, better solutions for that specific problem. Furthermore, the benchmarking method SIG uses is very easy to implement, and would not store any identifiable information.

SonarQube

SonarQube [24] is a well-established open-source continuous inspection suite by SonarSource. It analyzes both source code and version control usage. The software can be used in a centralized way, with a single instance providing metrics for multiple projects. It remains a collector of standard metrics and common style issues, however. It is not possible to generate a grade using SonarQube.

CodeClimate

CodeClimate provides Velocity, which performs analysis on the git usage of a project, and Quality, which assesses the quality of the code. Quality is available for free for small teams and open-source projects, but larger teams have to pay. Velocity is never free and prohibitively expensive [25]. Also, CodeClimate does not provide detailed feedback and can only provide suggestions based on a set of metrics.

2.4 Design choices

In this section, we explain what design choices have been made following the requirements gathering phase.

2.4.1 AuTA 1.0 as a base

We will continue to build upon the previous version of AuTA, because of the solid and extensible base. Other options include building upon an existing solution like JACK or starting from scratch. We believe we will be able to deliver a higher quality software product if we continue with AuTA 1.0, as we will have more time to focus on implementing the requested features rather than constructing a new framework to fit the new requirements. Therefore many of the technologies chosen for the previous project also apply to this project, the justification for which can be found in the architecture design report of last year which can be found in the code-measures repository.

AuTA 1.0 architecture

In AuTA 1.0, the system consists of three modules, the core, the worker, and the UI. The core is responsible for receiving submissions through its API endpoints. It also manages workers and sends these submissions to the workers to be analyzed. Finally, it generates a report and stores it in the database. A model of the system can be found in Appendix C.

The worker analyzes submissions. It receives a zip archive containing the submission to be analyzed. It then extracts the submission from the archive and runs its checks, before returning the results and removing the project files. Workers are automatically started and stopped by the core depending on the current load of the system, but they can optionally be manually linked to the core if necessary. A visualization of this flow can be seen in Figure E.1.

Finally, there is the UI. This provides a front-end to the core's API. It allows users, specifically TAs and instructors, to create or modify assignments and view results. Every interaction with the system using the UI happens through the API so no functionality is duplicated. AuTA 1.0's UI is hosted statically by the core.

JUMP

Communication between workers and the core in AuTA 1.0 was done using JUMP. JUMP was essentially created when we, during the development of AuTA 1.0, found that existing messaging libraries did not meet our needs for a simple, lightweight system for sending objects between the core and workers.

We investigated RabbitMQ [26] and ZeroMQ [27], being the options that best fit our needs. Both are more bloated and complex than required for our system. We concluded that creating our own simple messaging protocol would be the best solution.

In planning our changes for AuTA 2.0, we reviewed whether JUMP was still the best tool for this job. This resulted in our decision to keep JUMP as our messaging protocol, as alternatives are still not fit for our purpose. However, it needs to be extended with support for encrypted communication, in line with the requirements for GDPR compliance.

Modifications

At the moment, submissions can only be uploaded via the web interface (by uploading a .zip), using the API endpoint, or via the CPM integration. There are a number of ways to integrate this with GitLab Auto DevOps and Weblab. GitLab Auto DevOps can be customized by editing the .gitlab-ci.yml file to start custom Auto DevOps jobs. These jobs can start a Docker image and upload artifacts when complete.

One way to integrate this with AuTA's original architecture is by adding a job that starts an almost empty Docker container with a single script. This script would zip the repository, send this to the AuTA core, and poll the core until the job is complete. The core would then distribute this to available workers, which analyze the repository and return a results report. The core sends this to the waiting Docker instance, which returns the artifact (which can be a PDF, XML, JSON, or an HTML page for example) which can be viewed in GitLab.

Advantages:

- Minimal changes have to be made to the core, which will be faster to implement;
- Maintain 1 way to upload submissions to the core, which works the same for GitLab, WebLab, and CPM.

Disadvantages:

- More overhead: starting an empty Docker container to zip a file and send this to the core, only to be unpacked again in the worker might be slow for very large repositories.

A second way to integrate this with AuTA's architecture is by containerizing workers and running them in the Docker instance which is started by Auto DevOps. This containerized worker analyzes the repository and sends a results report to the core via the API. The GitLab instance hosted locally would be responsible for the management of containerized workers. A flow of this submission using GitLab can be seen in Figure E.2.

Advantages:

- GitLab's pipeline resource management can be used to manage worker containers;
- Less communication with the core, only results need to be sent back.

Disadvantages:

- Containerizing workers requires major changes to the architecture (especially the worker management section) and API;
- AuTA will need two worker modes: containerized for GitLab and possibly Weblab, and locally managed workers for CPM.

It is not possible to give an accurate estimate of the performance as this depends on too many variables (architectural changes, GitLab pipeline job management, etc.) We do not know which method will have better performance. This is why we have decided to start with the first integration method, as this will be simple to implement and test. We will work on the second method in parallel and compare performance when this is complete.

2.4.2 Java 11

AuTA will be upgraded to Java 11 (specifically, OpenJDK 11) because it is the next Long Term Support release after Java 8. We are upgrading from Java 10, thus the changes that will have to be made because of this are limited. All existing dependencies will be upgraded as well for compatibility and security reasons. AuTA will also not use any deprecated features of Java 11 so that it remains compatible with future releases of Java, which may bring architectural improvements such as better garbage collection.

2.4.3 Single-Sign On

AuTA will continue to use its own authentication system at its core, but Single-Sign On (SSO) support will be introduced with the help of the Spring SSO integration library. Initially, it will be implemented as a layer upon the existing infrastructure, with SSO serving only to request an AuTA authentication token. If additional security is required, a deeper integration will be added instead.

2.4.4 Benchmarking

An important requirement is that AuTA 2.0 is able to compare results for student's submissions with other students. This can be used to give the submission a score in a similar method as described in subsection 2.3.2. The current system does not couple a student's identity with the submission. Instead, each submission is given a unique id. To be able to show a student how they perform in comparison with others, the submission needs to be coupled to the student. If the student submits multiple submissions, AuTA 2.0 needs to compare only their last submission with the benchmark of other submissions.

2.4.5 Worker Management

In AuTA 1.0, workers are started locally by the core. Remote workers running on different machines can also be coupled to the core. Each worker can be started with a configurable number of threads. An issue with the old system is that each thread is regarded as a separate worker. This makes the management of workers more complex, as the core is responsible for managing all workers and their threads. The system could be improved by removing this responsibility from the core and moving this to the workers.

For improved scalability and easier worker deployment, the workers will be containerized in a Docker container. Using Docker Swarm we can manage the deployment of additional containers if the workload for the containers becomes too high. Each worker container will have a pool of worker threads and a job queue where jobs are stored until they can be analyzed by a thread. Java has built-in support for thread pool management using Executors and is able to create threads as they are required. Messages from the core will not go directly to the worker threads. Instead, a `ThreadManager` will be used to manage the creation of threads and distribution of jobs. See figure 2.1.

The old system often lead to a situation where many workers were spawned, each with a single thread. This is quite inefficient. The new method would reduce the memory footprint of workers, as more threads will be able to share resources.

Alternatively, a service like Kubernetes [28] could be used to dynamically scale the workers as demanded. An image containing a worker is replicated a number of times based on the CPU load of the existing workers also in the Kubernetes cluster. Kubernetes then acts as a load balancer endpoint which distributes the messages to the workers in a balanced way [29]. This requires additional research, however, as the platform is very complex.

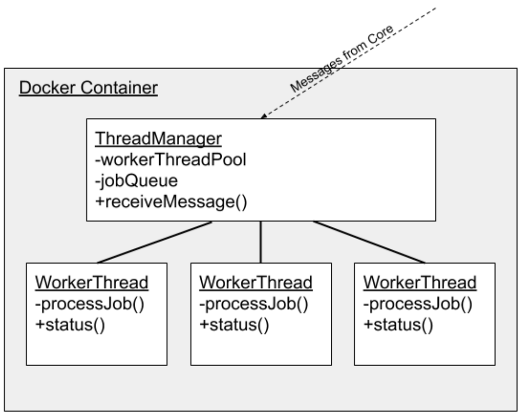


FIGURE 2.1: Diagram of new worker structure (non standard UML)

2.4.6 Metrics and code quality analysis

Test quality

As recommended by Maurício Aniche, we will use the Test Smells Repository [30] to analyze the quality of tests users provide for their Java and Python assignments.

This is an open source tool that is able to analyze many metrics used to find test smells.

Other metrics we can use to analyze code quality include the STREW-J metric suite presented by Nagappan et. al. [2]. These can be calculated using metrics included in the Chidamber and Kemerer Metric Suite. An example of a tool to do this is CK [31].

Code duplication

PMD, which is a static source code analyzer, contains a code-duplication detector (called CPD). This can be used as a command line tool and can produce XML file output and also supports C/C++, Java, and Python as input languages. PMD is mainly useful for larger project analysis, but CPD is also useful for smaller assignments.

Lizard

Lizard is a Python program capable of analyzing a very large set of languages. This program was chosen for its large language coverage and feature set, its ease of use and integration, and because of its very liberal license (MIT) allowing it to be included in an AGPL project . We will extend the features provided by Lizard to all languages AuTA will support to unify the back-end as much as possible.

Currently, Lizard is executed using a Python interpreter within the JVM instance (Jython), but due to the feature debt and concurrency limitations of the interpreter we may decide to switch over to the external executable CPython instead, if time permits. Further performance analysis is required, however.

Pylint

New in this version will be the inclusion of Pylint to lift Python support to the level of Java's. It supports all common quality checks for Python and is configurable so that it can be used for smaller projects and beginner Python users as well.

Pylint is licensed under the GNU General Public License v2, which is compatible with AuTA's AGPLv3.

Vue.js

The UI will be completely rewritten using the Vue.js library. In AuTA 1.0 the UI was rushed due to the failed collaboration with the group behind the fraud detection tool, which lead to severely unmaintainable code. Vue.js was chosen over React and AngularJS (its two main competitors) because it is lightweight and requires no additional tooling (unlike React) and has a stable history and good documentation (unlike AngularJS).

2.4.7 A note on data protection

As mentioned before in section 2.2.1, all dependencies will be audited in order to ensure that they do not export any protected information (such as submissions or their metadata) outside of the TU's internal network. Due to time concerns, this audit will take place after the initial research phase; therefore this list is subject to change, in case dependencies need to be replaced entirely because they cannot be made to conform to the law.

2.5 Future of Computer Science education

In this chapter, we will discuss what the future of computer science education might be with a functioning form of AuTA and related projects.

2.5.1 Labrador

AuTA is part of a bigger project: Labrador [32]. Labrador will be a single system containing AuTA, but also the system students use to enqueue for a timeslot during practicals (Queue) and a platform where students can upload their assignments (CPM). This system also encompasses automated plagiarism checks and an AI which can answer simple questions that a student might ask. This system will be used to reduce the time that a student has to wait to receive feedback and to reduce the pressure on TAs so that they have more time to answer more complex questions. The role of AuTA within this larger system will be to automatically check submissions from students and return submissions and/or a possible grade. It may also function as a 'gatekeeper' that students have to pass to be granted a slot in the queue for submission, depending on the course. To give a better overview of how we think Labrador will be developed, we provide two overviews, one simplified (Figure D.1) and one detailed (Figure D.2).

2.6 Conclusion

While many solutions for ensuring the quality and correctness of homework solutions exist, there are none that fully fit the requirements for customizability, scalability, and the quality of feedback, nor is it possible to adapt those existing solutions within a timely manner (with those meeting the requirements being multiple years out of date).

Generating feedback for larger projects based on standardized metrics is already possible using the existing solution, but work still needs to be done to generate useful feedback for students new to programming.

Chapter 3

Architecture

In this chapter, we will explain the decisions we made regarding the design of the tool for each of the parts of the tool.

3.1 Modular design

We split the tool up into three distinct parts: the core, the worker, and the user interface. The user interface is the main access to the tool for the administrator and instructors to set up the tool for usage. It is logically separate from the rest of the tool, as having a single programming interface makes it more maintainable. This also allows for development on the UI without having to update the rest of the tool, which makes the UI more maintainable.

The UI was further split into a frontend that was built using Vue.js, and a Javascript library - auta.js - that is responsible for the API calls to the core.

The core and the worker are separate modules mostly to allow for easy scaling. By separating the two we can connect multiple workers to the core and thus increase the throughput of the tool. It also allows us to run the core on a different server than the workers. The new major modules that have been added to AuTA are visualized in Figure 3.1. For a more complete diagram, see figure C.2. For comparison, The component diagram of the first version of AuTA can be seen in figure C.1.

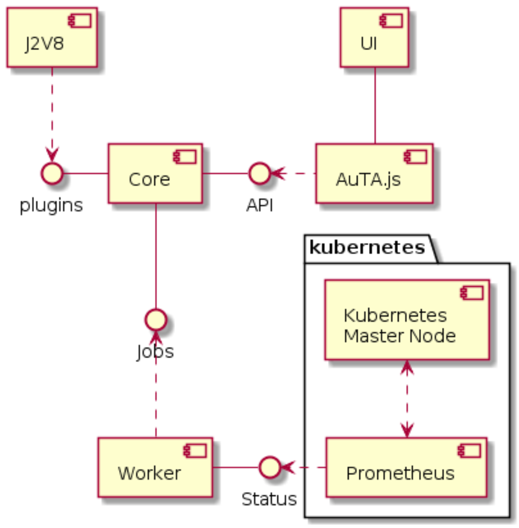


FIGURE 3.1: The new AuTA structure. (Uses PlantUML syntax [33])

3.2 Kubernetes

To make sure we run our system (and student code) in an isolated environment and make use of our modular design, we chose to use Kubernetes.

During the research phase, we had the idea of using a `ThreadManager` to manually balance the load on the workers (see section 2.4.5). With the realization that AuTA 2.0 will be run on multiple physical machines and that workers will be containerized, we decided to not implement the `ThreadManager`, as it would require quite an advanced system that we would have to write ourselves so we instead settled on using an existing container orchestration platform.

When designing the structure of AuTA 2.0, we wanted to use a container orchestration platform to allow us to easily manage a large number of worker containers. After some research, we settled on two main options: Kubernetes and Docker Swarm. We decided on Kubernetes since, besides it being a very popular tool for container orchestration, it allows for automatic scaling of containers depending on the load of the cluster.

Kubernetes' Horizontal Pod Autoscaling thus eliminates the need to handle worker load balancing ourselves. This meant that the simple load balancer that was created for the first version of AuTA could be removed, as Kubernetes now handles this task. Kubernetes' autoscaling is very flexible and allows a large number of parameters to be configured, including, for instance, the minimum and maximum number of workers and at what percentage of busy workers new containers should be started.

To allow us to use Kubernetes' autoscaling with our own resource metrics, namely the percentage of busy workers, we use Prometheus. Prometheus is used for application monitoring and alerting, but in AuTA it serves to allow Kubernetes to use our custom metrics for autoscaling. Workers expose their status, which is either busy or available, through a simple HTTP server. Prometheus then collects these metrics, passing them on to Kubernetes which starts or stops new containers based on this information.

3.3 REST API

External tools can communicate with the core through the REST API. We chose to implement the core as a RESTful web service. After reviewing our options we chose this as the most logical option. Communication with the core should be as simple and intuitive as possible, which is one of the major advantages of a REST API compared to, for instance, a bare TCP socket.

3.4 AuTA.js

To allow an external service to easily communicate with the core, we have developed AuTA.js. This is a JavaScript library with which you can easily call the API of the core. A separate library was created as opposed to integrating it within the UI as now we allow for a possible different services, including user interfaces, to be built around it without having to recreate the interface with the core. This is also in anticipation of the integration with Labrador, as it is not yet certain how that UI will be given form.

3.5 Core Plugins

AuTA 2.0 features a plugin system. Using this system, external tools can be loaded by the core to extend its functionality. Currently, the only use for it is the J2V8 plugin, but the system is set up to allow for future additions. Plugins are loaded from disk early in the core's bootstrapping procedure by scanning a directory for compatible jars. Plugins can be of one or more *plugin classes*. These classes determine how the plugin is going to be used by the core. At the time of writing only one plugin class is supported, namely `AuTA-SRF`, which is for plugins providing scripting contexts and implemented by the aforementioned J2V8 plugin. Plugins with classes unknown to the core are not loaded at all. Plugins must declare the plugin classes they implement in the `META-INF` directory. Every plugin class is free to define further semantics; in the case of J2V8, the activator class defined in the `AuTA-SRF` file is the factory that will be building script execution contexts.

3.5.1 J2V8

As mentioned above, J2V8 [34] is currently the only plugin that is currently used in AuTA. J2V8 is a library that provides bindings for Google's V8 engine [35]. Our previous solution for running scripts involved using MongoDB eval statements to run scripts in the database. This was, however, unworkably slow as submissions become larger. In addition, the eval function that was used has been deprecated.

J2V8 was chosen over alternatives such as Rhino [36] and Nashorn [37] because it is still being actively developed and has better performance than the alternatives.

Due to the license used (Eclipse Public License), the only way to use this library is if the end-users of the software install this library themselves as a plugin. It is not allowed to be included in AuTA by default. A new repository was created to convert the J2V8 library into a plugin, which can be included as a .jar using the system described above.

3.6 SAML Single-Sign On

In addition to AuTA's own authentication methods, the core also exposes a SAML 2.0 integration that allows users to authenticate using a Single-Sign On identity provider.

AuTA is of itself SAML 2.0-compatible as a Service Provider [38][39]. By default, the system is configured to enforce very strict security standards [40]:

- All requests are cryptographically signed;
- All responses have their signatures checked against the identity provider's public key;

- Authentication requests are valid for a limited amount of time: by default, requests expire after 30 seconds;
- Accepting new users and logging in may be (separately) disabled per identity provider.

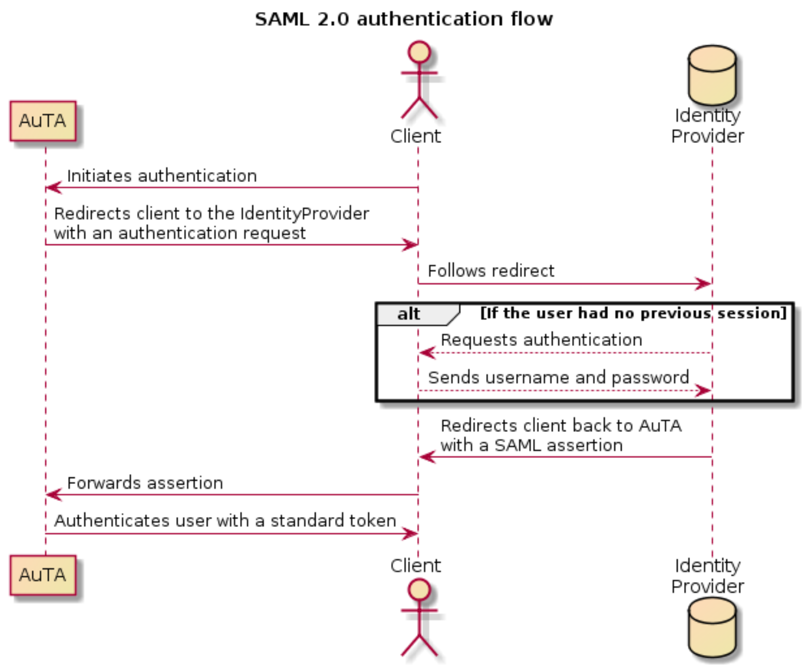

SAML 2.0 authentication flow
AuTA
Client
Identity Provider
Initiates authentication
Redirects client to the IdentityProvider with an authentication request
Follows redirect
alt
[If the user had no previous session]
Requests authentication
Sends username and password
Redirects client back to AuTA with a SAML assertion
Forwards assertion
Authenticates user with a standard token
AuTA
Client
Identity Provider

FIGURE 3.2: AuTA SAML 2.0 SSO authentication flow. (Uses PlantUML syntax [33])

After successfully authenticating, the user is granted a standard AuTA authentication token the user can use for the REST API requests and thus the user interface. This token is valid for a limited amount of time after which re-authentication with the identity provider is necessary. If no existing account is found for the credentials in the assertion, then a new account is transparently registered (as long as the configuration allows it for the identity provider).

The internal user's username, which must be unique, is generated based on the identity provider's entity ID with the assertion's NameID as a fragment. This ensures that no existing user can be accessed using a different identity provider. Because this username is a URI, this is no longer suitable for display and so the assertion is searched for a myriad of standardized attributes that could provide a display name [41][42][43]. Only if no value is found, then the display name defaults to the name ID.

3.7 Trampoline

One of our client's requirements for the tool was an integration with GitLab Auto DevOps. In section 2.4.1, a trade-off was made between:

1. Starting a Docker container that zips the repository, sends it to the core and waits for a response;

2. Starting a containerized worker that runs an analysis on the job, sends the results to the core and generates a report.

Option 1 was chosen due to easier implementation. In a new repository, we created a Python script called Trampoline that does exactly what is described above. In addition, it is able to read the configuration file in the root directory of the project. This configuration file may contain a list of glob-style patterns that describe files that do not need to be sent to the core for analysis. This turned out to be necessary during tests with our own project - the .git version control folder was becoming so large that the server began to reject submissions.

See Appendix section C.3 for an activity diagram that describes what Trampoline does.

3.8 JUMP

JUMP is the communication layer between the core and its workers. This was created for the first version of AuTA, and updated with encryption in AuTA 2.0. The encryption standard that was chosen is the advanced encryption standard (AES) [44] as this is a widely available, well-supported encryption scheme that is generally agreed to be secure enough to protect sensitive information [45].

The JUMP protocol consists of two layers: the encrypted layer and the unencrypted layer. The unencrypted layer is very simple, connecting to another JUMP instance with a socket connection. JUMP simply transfers a message as the byte array of a string, with a header and footer, 32 bits network order each, describing the size of the message.

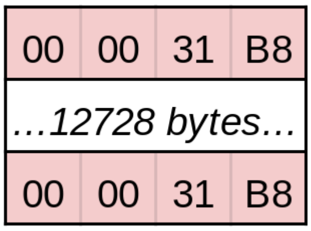


FIGURE 3.3: A JUMP message

Encrypted JUMP or JUMPS is a layer on top of unencrypted JUMP, which means that the message header and footer are not encrypted. The core is responsible for sending workers a request to use encryption. The core has a public-private key pair, sending its public key along with the control message to enable encryption. The worker then generates a random AES key, encrypting it with the core's public key and sending it back to the core. From the point where it sends the new secret key, it expects subsequent messages to be using encrypted JUMP. Upon receiving the response from the worker, the core decrypts the new secret key with its own private key, using the secret key in all new messages to that worker. See figure 3.4 for a sequence diagram that illustrates the key exchange.

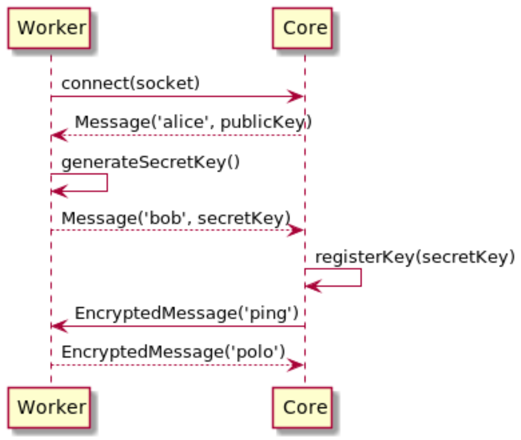


FIGURE 3.4: Sequence diagram showing key exchange flow between workers and core. (Uses PlantUML syntax [33])

Every worker uses a different, randomly generated secret key. The keys are generated at startup and different even when restarting workers. The core uses a public-private key pair to allow workers to send their secret key without the risk of it being intercepted.

The initialization vector used when encrypting messages is always set to zero, which could lead to a vulnerability if enough messages could be intercepted. However, this would require a large number of messages [46] and even if one of these secret keys could be intercepted, a valid API key is still required to access any student code. These keys are not transmitted over JUMP(S) and instead already specified at the startup of a worker. Furthermore, it is expected that the workers and the core communicate within a private network, which makes intercepting the messages even more difficult.

3.9 Worker - Core communication

The core communicates with the core using a number of control messages, which are communicated via JUMPS. The core sends a 'ping' message to the worker at a fixed interval, usually around 30 seconds. The worker is expected to respond with 'polo'. If it does not respond in time, the worker is culled.

Other control messages are used to cancel jobs, kill workers, notify the core that the worker is busy, or initiate encrypted JUMP.

3.10 Entity structure

For the storage of our results, we use a very similar structure to the structure described by SIG [21]. The first version of AuTA used a hash map that mapped metric names to a result object. The advantage of this schema-less storage method is an increase in flexibility in what can be stored. However, a lack of a fixed schema causes a lot of issues when trying to do anything with the results object. This was especially apparent when trying to aggregate metrics for benchmarking.

This is why we decided to refactor the results storage completely. We decided to build a tree-based structure of entities. The root entity is the project entity. Each entity can have child entities that are a different level (module, file, class, method, field).

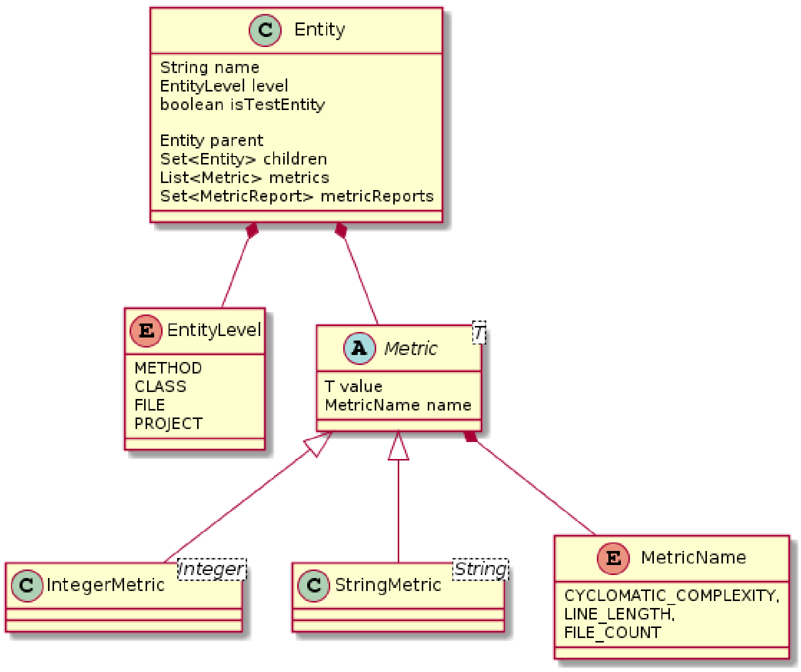


FIGURE 3.5: Entity structure UML. (Uses PlantUML syntax [33])

By using this structure we also preserve the structure of the project; this makes it easier to generate a report which is easily understandable for the students. When the workers receive a job, it first builds a partial entity structure. This structure consists of a project entity with all file entities as children. The name of the file entity is the path to the file, which can be later used in the report to identify the correct file where a problem lies.

After creating this initial structure, the partially created project entity is passed to the analyzers. These analyzers continue to build the entity structure and add metrics to the entities. See section 3.12 for more information about what the analyzers do.

3.11 Metric calculation

The worker is the component where actual code analysis will take place. The worker starts by downloading the archive associated with the job. This archive is unpacked and then used to build a project entity structure. Only files with allowed extensions (e.g. .java for Java assignments) are added to the entity tree. The entity tree is built down to a depth of file entities, as we did not have time to implement full project parsing before analysis. See section 9.5 for a discussion on how this can be improved.

In the job splitter, we also mark file entities that contain tests as such. Similarly to the method described by Alves et. al. (2009) [47], we do this by detecting any imports of the JUnit library. As Alves et. al. mentioned, the disadvantage of this method is that test helper classes cannot be detected. This could be improved by detecting more libraries that perform tests, such as NUnit, TestNG, or Mockito. This would have to be repeated for different languages. Another possible solution, involving machine learning, is described in section 9.6.

This project entity is then added to a job, which also contains a project entity, the language of the job, various options, and a path to the unzipped directory. This job is then passed to the Checker, which is responsible for gathering the correct analyzers needed to calculate the requested metrics associated with the job.

Using the metric names that are passed along with the job, the Checker creates a list of analyzers that need to be called. These analyzers are then run and the results are combined and sent back to the worker's job runner. The results are then returned to the core. See figure 3.6 for a sequence diagram explaining the message flow throughout the worker.

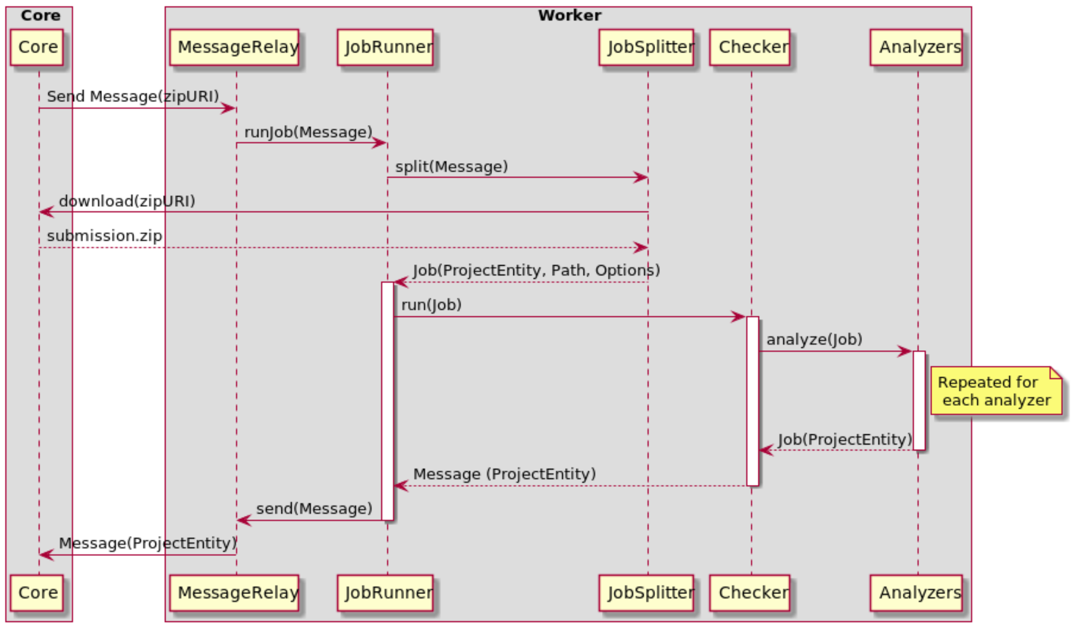


FIGURE 3.6: Sequence diagram for submission message processing. (Uses PlantUML syntax [33])

3.12 Analyzers

An analyzer is any class that can perform an analysis on a file entity or a job. Job analyzers are required for tools that perform analysis over the entire project (e.g. duplicate code block detector, file counter, Docker container runner). These analyzers are only run once for the entire project. Entity analyzers (e.g. Lizard, CK, Radon) run once for every file. Aggregated analyzers run after all job analyzers and entity analyzers are complete. They use the metrics that have been added by other analyzers. See figure C.4 for a class diagram of the analyzers.

In general, an analyzer calls a tool that produces results in some format (JSON, XML, plain text). These results are converted to metrics and in some cases entities. For example, Lizard (a Python tool) is used to parse most languages and add class, method, and field entities to the correct file entity. Lizard also calculates metrics, that are added to the correct entity (file or method). See figure 3.7 for an example sequence diagram.

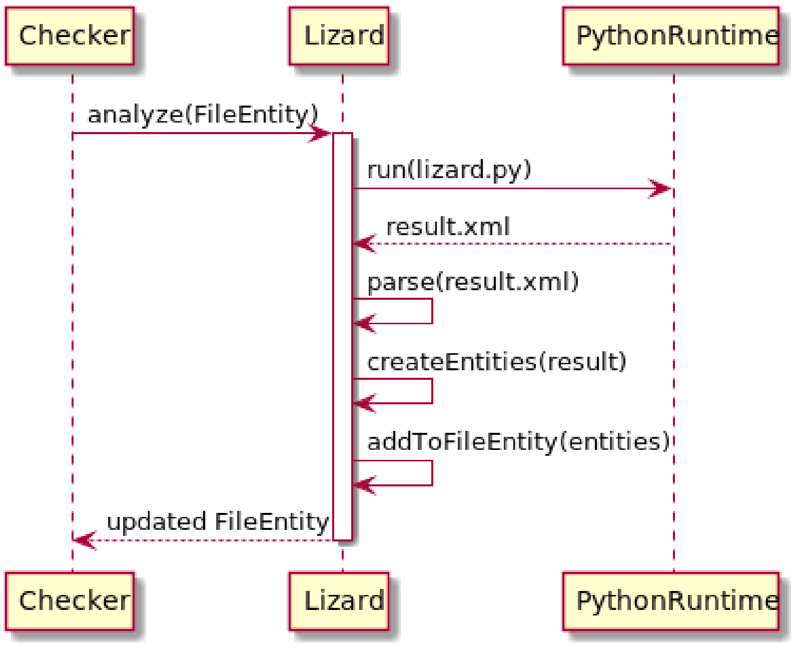


FIGURE 3.7: Sequence diagram example for analyzer (Lizard). (Uses PlantUML syntax [33])

New tools are relatively easy to add to the workers, simply by creating a new class that extends a `JobAnalyzer`, `EntityAnalyzer` or `AggregatedAnalyzer`. When the worker starts up, the checker automatically creates a list of analyzers by parsing the Analyzer file in the META-INF/services folder of the jar. This file contains a list of all implementations of the Analyzer interface. When the Checker receives a job with metrics, it looks up which analyzers are able to calculate any metrics for the job's language. The list is then reduced to the minimum number of analyzers required to calculate all required metrics.

3.12.1 Dynamic analysis

A special case is the Docker analyzer. It allows whoever is creating the assignment to specify a Dockerfile which should be run. The student code can be accessed inside a volume mounted to the container in `/var/auta/submission`. Currently, the script allows the logs to be parsed and to set conditions as to when it should fail or warn.

3.13 Metric scripts

Each metric has its own passing scripts. These scripts are used to analyze the retrieved metrics from the analyzers and determine if they fall within the given boundaries of the assignment. Depending on the analyzer, the metrics returned can be simple primitives or complex objects. The script is responsible for assessing these values and return notes, which can be tips, warnings, or outright failures. To give instructors the most flexibility the scripts can be altered when creating the assignment. This way the instructor can determine their own threshold or even create more specific scripts that suit the needs of their assignment.

Javascript is used to generate warnings, failures or tips for students. Using the user interface, teaching assistants or instructors can use our pre-programmed scripts, adjust thresholds or program their own custom scripts. See figure K.6 in the appendix for an example script.

Internally, the J2V8 plugin (see section 3.5.1) is used to compile, execute and return the script's output. Scripts are run for each metric in every entity. This output is then added to the metric object (which is contained in the entity) and saved in the database.

3.14 Report

After the analyzers are finished, a report is generated. The report consists of the notes that were created by the analyzers. The report retrieves the tips for the metrics as soon as there is a warning or a failure for that metric. A list of entities where the warning or failure occurred is displayed under the tip to easily identify where the problems occur. Further down the report the actual warning and failures are displayed to give detailed info as to why the metric failed. See appendix chapter J for examples of reports.

Another type of report that is generated is a histogram that shows the relative weight of code with a specific metric value. These histograms are a very useful visual method for TAs and students to identify methods, classes or files that may contain problems. The histograms are produced using the method described in section 2.3.2 (excluding the system comparison steps). See figure J.3 in the appendix for an example of a histogram for cyclomatic complexity.

This aggregation of metrics to generate reports is very fast, which is why we have decided not to cache the reports on the server. Instead, an endpoint has been created where reports can be requested on-demand.

An overview of how the core handles a submission from initially receiving it to creating the final report can be found in figure 3.8.

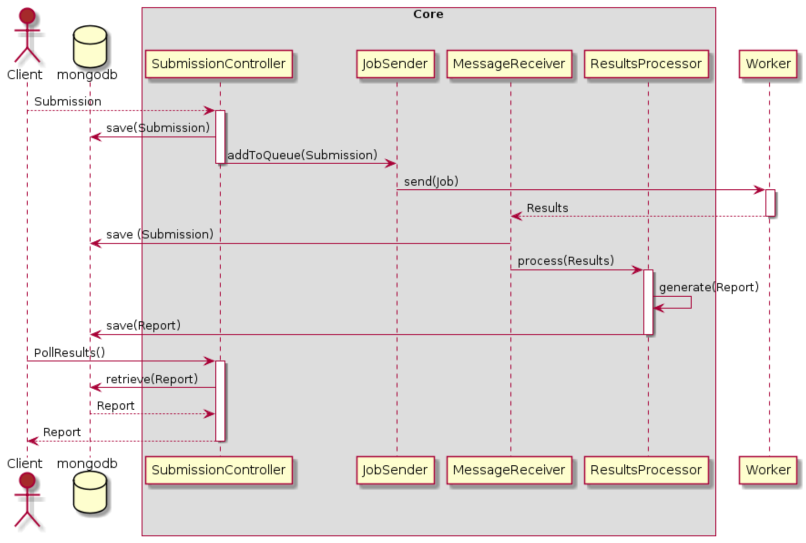


FIGURE 3.8: Sequence diagram for the core. (Uses PlantUML syntax [33])

3.15 Feedback generation

We have two ways of returning feedback to the student. The first is through a docker container. The instructor can use this docker container as they wish and return feedback themselves. The second method is through the metric calculations. A tip can be added to each metric that will be returned when the metric is violated. This tip will most likely contain a hint as to how the code might be improved.

3.16 Method names

AuTA uses a variety of tools as analysis backends. Unfortunately, every tool has its own format for reporting method names. To enable accurate reports and benchmarking, these results have to be merged. Figure C.5 in the Appendix shows how this process of merging works.

We used ANTLR grammars to parse the different results that tools gave. Below is a description of the requirements for method names to be used internally and a list of the tools used as backends and their outputs.

3.16.1 Internal formats

Because the tool supports multiple languages, the requirements are defined per-language. Name generators must be aware of the language they are generating names for.

3.16.1.1 Java

Any Java name used internally must conform to the following requirements:

- **The name is package-unqualified**, i.e. excluding the leading package. Most type usages follow this format and scanning a file's imports and package environment for the corresponding package name would be significant work for minimal gain. This introduces an edge-case where a method m_1 overloading m_2 exists accepting a type that has the same top-level class name as m_2 but from a different package. Such usage is extremely rare.
- **Parent classes *are* included in the type name.** This potentially introduces merging bugs when the type's parent class looks like a package (that is, the name is not capitalized), but this is in violation of the language's fundamental naming conventions so the submitter was pretty much asking for it.
- **The signature has been erased of any type parameters.** The Java Language Specification forbids overloading on type parameters due to erasure so this allows AuTA to use tools that do not return any generic information, such as older patches of Lizard.
- **Other modifiers are ignored.** This includes the `final` keyword and any annotations.
- `var` **as a variable name is allowed** for backwards compatibility, but a warning may be issued.

3.16.2 Tool Outputs

3.16.2.1 Lizard

Java
Lizard was patched in order to include parameter information in its reports as the stock version outputs the name with an ellipsis instead of parameters. The first version of AuTA used Lizard v1.14.10 and included the generated "long name", which was the full method signature but without any annotation attributes and mangled array types (only the closing square bracket was included). Newer versions of AuTA use a patched Lizard v1.16.3 that outputs the method's package, name, and parameters separately. Returned method signatures are mostly copied as-is, with some spaces added between tokens where permitted. Lizard replaces dots with double colons if they delimit packages. It is unknown if Lizard can detect child types of classes instead of packages, but it appears possible (we have not found the code that performs this check, however).

Python
With the new patch, Lizard behaves the same for Python. Because the Python grammar for function definitions is very simple [48], no dedicated parser is necessary.

3.16.2.2 CK

Java
CK outputs methods as `name/arity[type,type,...]`. Types are (seemingly) randomly package-qualified and include generics. A specialized parser is present in the worker to parse such results, interpreting the types as Java types. The names

are generated using Eclipse's JDT Name::getFullyQualifiedName(). CK does not include the name of the parameter.

3.16.2.3 Test Smells

Java
Test Smells does not output parameters, which makes qualified name generation very difficult.

3.17 Language detection

Currently, AuTA is able to detect the language of submissions. It does so by running Github's Linguist [49] on the student code. We have investigated several tools for the detection of languages from code, but Linguist proved to be the best available tool since it is capable of detecting multiple languages and provides output that is easy to parse. Linguist was written in Ruby and can be run by executing the github-linguist command from the student code's directory. It then outputs a breakdown of each language found in the code, along with the percentage of lines written in that language. AuTA finds the student submission's language by taking the language that has the largest amount of lines.

3.18 Suppressions

It is possible to suppress some files from analysis. To do this, add a `.auta.yml` file to the root of the project with the following format:

```
suppressions:
    - path: worker/src/test/resources/**
      justification: Test resources are intentionally bad to test analyzers
```

Under the `suppressions` key, an array of suppression specifications follows. Each specification consists of a path and a justification. Paths are Ant-style matchers [50], so advanced patterns are possible. Justifications are strings that are passed back along with the analysis results to remind the teaching assistant of any suppressions (or even failing the submission if suppressions are forbidden). All parts of a suppression specification are mandatory.

3.19 Security

Security is a very important facet of AuTA. This section describes how the system is designed and works in daily operation.

3.19.1 Tokens

Most API endpoints require authentication, which is done using the `Authorization` header with type `AutaToken`. Tokens vary in size (this is configurable), but generally exist of a large number of bytes generated by a cryptographically secure random number generator. The bytes are expressed in capitalized hexadecimal numbers concatenated together without separators:

```
1A9E52B3B76FD96C5D3D9C215941B43D790957C6B6D8319CA8B1622EEFA2D371
6FFD7DDAD3F85664C31B5C1078CA5CA40C00036E73CE9A2923101439D30F384D
```

Tokens are unique and identify a user (a user may have multiple tokens), to ensure that the tokens are unique a sufficiently high amount of bytes has to be generated. The default generator, the `TokenGenerator`, is unit-tested for uniqueness, with the lowest guarantee starting at 128 bits (16 bytes). The default setting is 512 bits (64 bytes), an example is shown above. The class is tested for support up to 65536 bits. The default size of 512 bits was chosen because it is practically impossible to get a collision, while the size of the string remains manageable.

3.19.2 Management API

Users can be managed through the API. New users can be registered (`/user/register`), existing users can request a token by authenticating using their username and password (`/user/login`), authenticated users can invalidate their tokens using the (`/user/logout`) endpoint.

3.19.3 Token authentication

The token authentication filter sits in the request-response pipeline and extracts the authorization token from the request before the general Spring request security filter applies the security configuration rules. This is the class that communicates with the database and attempts to find the user associated with the given token. If this fails, the user is set to anonymous (and, for the majority of the endpoints, the request will fail).

3.19.4 CORS Patch Filter

Due to a bug in the Spring Security Cross-Origin Resource Sharing implementation, preflight `OPTIONS` requests are subject to the same access restrictions as any other kind of request. However, because the browser is unable to send any `Authorization` header along with the preflight request, all CORS requests against secured endpoints fail. This filter detects such preflight requests and grants a virtual authorization with the `CORS` role, which is allowed to access all endpoints. Security is not impacted, however, because the filter is only applied to OPTIONS requests, which do not return any application information (just headers). Due to *another* bug in Spring, if an error is generated somewhere down the chain, no CORS headers are added to the response. This makes the browser reject such errors, meaning that the application can no longer handle them. This filter detects such cases and adds the appropriate headers anyway. This only happens if the origin is an allowed origin.

3.19.5 Security Agent

The security agent performs security checks at regular intervals. It raises concerns about the following aspects:

Severity	**Cause**
critical	The authentication database is accessible by everyone
critical	The AuTA core runs as the superuser
warning	The default accounts still have their default passwords
notice	The authentication database is executable

TABLE 3.1: Security agent concerns

By default, these checks run at startup and every 30 minutes and the agent may take action to ensure that the system remains secure. By default, this means that every concern gets logged and the system is shut down once a critical error is detected.

Chapter 4

Tools

In this chapter, we will describe the third-party tools that we used and Incorporated into our project. We will explain what metrics the tool generates and under which license the tool is published.

4.1 Test Smells Detector

The test smell detector detects test smells for java unit tests. The test smell detector was originally developed by the faculty and students of the Department of Software Engineering of Rochester Institute of Technology, New York [51]. However, the tool has some flaws in the code which rendered it useless. The research that went into the tool, however, was very extensive, thus we decided to make use of it after all. We forked the repository and made some changes where necessary. The tool is licensed under GPLv3 and is therefore compatible with our own tool. See appendix section H.1 for the list of test smells the tool produces.

4.2 Lizard

Lizard [6] is a Python program capable of analyzing a very large set of languages: `C`, `C`, `Java`, `C#`, `JavaScript`, `ObjectiveC`, `Swift`, `Python`, `Ruby`, `TTCN3`, `PHP`, `Scala`, `GDScript`, `Golang` and `Lua`. It has an MIT license and is therefore compatible with our tool. Lizard supports the following metrics:

- Lines of code - The number of lines that are in each file.
- Cyclomatic complexity - A measurement for how complex a piece of code is.
- Parameter count of functions - The number of parameters per function.

A patched version of this tool is used to generate qualified names for Python and Java, and to calculate the metrics listed above.

4.3 QDox

QDox [52] is a Java parser that we use to manually check some metrics in java code if we could not find a tool that could do it for us. QDox is licensed under Apache 2.0

and is thus compatible with our tool. The Javadoc analysis depends on QDox, for example.

4.4 PlantUML

PlantUML [33] is a tool that generates a UML diagram when given a correct puml file. This tool is licensed under GPLv3 and therefore compatible with our tool. We use this tool to generate UML diagrams for Java projects.

4.5 CK

CK [31] is a tool that analyzes class and method level metrics for Java projects. The tool is licensed under the Apache 2.0 license and is thus compatible with our tool. The tool has a wide range of metrics that it analyzes, which can be found in Appendix section H.2.

4.6 Radon

Radon [7] is tool that calculates metrics for Python code. The tool is licensed under the MIT license and is thus compatible with our tool. We use this tool to calculate the maintainability index of a file. The tool is capable of calculating more metrics, but these are already covered by another tool (Lizard) which is more efficient in doing so.

4.7 PyLint

PyLint [53] is "a Python static code analysis tool that looks for programming errors, helps to enforce a coding standard, sniffs for code smells and offers simple refactoring suggestions". It is only compatible with Python and offers a wide range of cases it can detect and highlight. The tool is licensed under GPL-2.0 which means it is compatible with our tool.

4.8 Profanity Checker

Profanity Checker [54] is a Python tool that checks for profanity in source files. It is licensed under MIT and is thus compatible with our tool. This profanity checker is a python library that uses a Support Vector Machine, trained on a large collection of human-labeled data from Twitter and Wikipedia. [55] Zhou's profanity checker fits our tool perfectly since not relying on word lists reduces the possibility of accidentally marking clean words that contain profane words as profanity [56]. It also allows the checker to pick up on variations of words that a word list based checker would not pick up on since that specific variation does not appear on the list. The checker, however, is not infallible. Without additional training data, it will, for instance, only work on English profanity. But at the moment, using this tool our checker is able to detect some of the most common profanities in student code, teaching the students to behave in a more professional manner.

Chapter 5

Performance

In this chapter, we will discuss the performance of the tool after performing stress tests.

5.1 Method

The performance of the tool was analyzed by performing stress-tests. These tests were performed using a machine with the following specifications:

- Intel Core i7-8750H 2.2 GHz CPU with 6 cores.
- Hynix 1x 16 GB DDR4 2666 MHz RAM.
- SAMSUNG MZVLB512HAJQ-000H1 512 GB SSD

The core and workers were started without a Docker container. This meant that there was no autoscaling using Kubernetes, as this would have complicated testing. Manually starting workers also keeps the number of workers constant.

Between tests with different numbers of workers, or after workers had been idle for some time the JVM was warmed up by sending 20 submissions to the core (which takes approximately 40 seconds).

Submissions were sent using a Python stress testing script, which is simply multi-threaded Trampoline with a GUI. A variable number of files can be sent to the core simultaneously, which simulates the load the core might have to handle close to a programming assignment deadline. Up to 200 simultaneous submissions were tested, after which there was no perceivable change in effective lines of code analyzed per second.

Logging was set to info-level logs, to reduce I/O which slows the system down considerably (10% speed decrease when using trace level logging).

The total runtime was recorded for a variable number of workers with different numbers of submissions. 1, 2, 5, 10, 20, 25, 50, 100, and 200 submissions were sent to 1, 2, 4, and 8 workers. 200 submissions were sent to 3, 5, 6, and 7 workers. The submissions that were sent were a random selection of 5 different solutions for the Object Oriented Programming course's fifth assignment. These solution sizes ranged from 650-1400 ELOC. The mean ELOC is 1053, with a standard deviation of 372 ELOC. All available metrics for Java were calculated, with the exception of Test Smells and Docker Logs. We think this assignment is a representative submission that might be uploaded to the tool when it is first put in use. The results can be found in appendix N.

Figure N.2 clearly shows that the average speed when analyzing 200 submissions increases linearly up to a certain number of workers, after which the analysis rate stays relatively constant at around 1400 ELOC/s. Figure N.3 shows that running 8 workers is actually slightly slower on average than running 4 workers, but both have very similar performance.

Tests were also done with larger projects (our entire code base, which is approximately 29400 ELOC.) With 4 workers and 8 simultaneous submissions, the total analysis time was 180 s. The average speed was approximately 1300 ELOC/s.

We also profiled the following areas for both the tests with small and large submissions

Submission size	**Report generation**	**MongoDB insertion**	**HTML report generation**	**Benchmarking**	**Total worker analysis time**
Small	800-1500 ms	50-120 ms	6 ms	6 ms	2500-4000 ms
Large	15000-30000 ms	1500-3000 ms	1000 ms	2000 ms	81000-85000 ms

TABLE 5.1: Profiling comparison for small and large submissions

5.2 Discussion

There is clearly a bottleneck in the tool that is preventing the system from scaling effectively beyond 4 workers (on the machine the stress tests were performed on). The speed decrease that occurred when adding more than 4 workers is probably due to an increased overall system load (memory usage, all cores in use) which causes the performance of the bottleneck to decrease. Memory usage by the core did not exceed 1 GB and each worker used around 400 MB. Maximum IO Read/Write was 3 MB/s. We suspect that report generation in the core is the bottleneck, not the workers. The system is designed to be able to run workers on a separate machine to the core; scaling the workers should not be a problem.

A possible solution to report generation being a bottleneck on the core could be delegating report generation to workers. This would allow report generation to scale on several machines and likely decrease the analysis time per submission.

We think that this performance is acceptable for the tool's use case. If a minimum of 4 workers is always kept alive, students will have to wait no longer than 150 seconds if 200 students submit at the exact same time. If 50 students submit their code at the same time, this wait time is reduced to 40 s. This is shorter than most builds take.

For larger projects, it is acceptable to have to wait longer. Build pipelines for projects of this scale generally take more than 10 minutes, so the wait time would not be increased as this stage can run in parallel on the build server.

Chapter 6

Process

In this chapter, we will reflect on the process of the project. We split it up in a research and a production phase as the processes differ significantly.

6.1 Research phase

In Chapter 2 we describe the process of the first two weeks of the project. In short, we interviewed multiple stakeholders and our client to create a prioritized list of requirements that we would refer to throughout the entire project. A literature review was used to gain an understanding of one of the more difficult parts of our project, which involved producing useful feedback for students. New tools and metrics were found that could be integrated with our product.

6.2 Production phase and challenges

In this section, we give a general outline of the development process and in what order features were introduced. We then describe the main challenges that we encountered during this project, and how we managed them.

6.2.1 Outline

After completion of the research phase, we started on the development of the actual tool. We started by following the project plan that we had created during the research phase. However, we quickly started to deviate from this plan as some tasks took longer than expected and others were finished much sooner.

The production process began by updating the first version of AuTA and designing and developing a method to integrate with GitLab Auto DevOps using Trampoline (which is described in section 3.7). This was done as a part of our continuous integration testing strategy, see section 6.2.4. This integration was easier than expected, as the GitLab DevOps system is very well documented and easy to integrate with.

Our next step was redesigning the way we managed workers. As described in sections 2.4.5 and 3.2, we chose to use Kubernetes to manage the lifecycle of workers based on system load. Our initial idea was to run AuTA permanently on a VM on the TU Delft servers. However, the Kubernetes master node needs to run on a different (virtual) machine than the worker pods. Requesting a second VM could take a few weeks, which we thought was too long. Instead, we decided to deploy it on one of our own servers, and buy a Raspberry Pi 3 which would run the Kubernetes master node. A detailed manual (Appendix G) was created explaining how to set up Kubernetes, which was required by our client.

Next, the old UI was ported to Vue.js and split into a front-end and auta.js (see section 3.4). Porting to Vue.js turned out to be a very good design decision from a maintainability and development perspective. It was very easy to add new pages to the new UI compared to the old UI, and the code size decreased considerably while increasing quality.

Updating the report generation method, which is described in section 3.13, was very easy and took less time than expected.

After the midterm meeting, we started focusing on delivering a functional tool and less on implementing new features. We therefore mainly focused on fixing bugs that were introduced during the first stage of the production phase and adding the last new features which are necessary to deliver a tool that is immediately functional when the project ends.

To keep everyone updated during the production phase we had a meeting with our coach every Friday. We also sent an e-mail to the client and our coach explaining what we did that week so that everyone was aware of how far along we were with the project. We also tried to give a demo each week to show the added functionality.

6.2.2 Challenges

6.2.2.1 Revisiting old code

When starting work on AuTA 2.0, we found ourselves working on code that we had written a year earlier, as we also worked on this project during the context project. Even though we have not had any programming courses in the meantime we were still dissatisfied with parts of the old codebase. This is a valuable learning experience as you find out first hand as to why writing maintainable code is important and what makes code more or less maintainable.

6.2.2.2 Tool output merging

New metrics and tools, which are described in chapter 4, were added throughout the entire development process. However, we only realized that the merging of different tools results was going wrong when the report generation method was updated. Designing a way to merge the results of each tool, such that the entity names were matched, proved to be a major challenge in our project. A working solution took until Week 8 to be completed. Although AuTA did produce a report before this, the report was subtly incorrect until this had been solved. See section 3.16 for a more detailed description of the problem and solution.

6.2.2.3 Single Sign On

Single Sign-On integration was another major, complex challenge that we faced during the production phase. We had expected this to be easy to do, as we expected to be able to use OAuth2 which is well-supported by Spring. The documentation that was sent to us by Frans Broos, head of ICT of the EEMCS faculty explains that an application needs to be made SAML compliant in order to be compliant with the TU Delft SSO. Our SSO integration makes use of SAML 2.0 using the OpenSAML 3 library provided by the Shibboleth Project [57].

Initially we planned on using Spring extensions to handle the SAML flow, but these were outdated [58] (as they used OpenSAML 2, which reached end-of-life on July 31, 2016 [59]). Other extensions like Shibboleth Service Provider require integrating with the Apache or IIS web servers [60], which is a layer outside the control of AuTA itself, which is why we opted to integrate OpenSAML 3 manually.

This was a very large undertaking, however, we believed that using an outdated implementation of SAML flow was a bad idea due to security concerns.

As of writing, the user interface for SAML SSO is yet to be completed. Because the SAML 2.0 specification is very complex, the OpenSAML 3 documentation is very lacking, and authentication is a core component deeply ingrained in the system, the development of the integration took longer than expected.

In the future, SSO may be required for different universities (e.g., VU Amsterdam). Our solution makes use of the standardized attributes also used by SURFconext [61]. SURFconext is the authentication and authorization service used by many Dutch educational institutions [62]. By using standardized attributes, other institutions can also easily integrate their SSO service with AuTA, as AuTA can handle multiple identity providers.

6.2.2.4 Failing Build Servers

A troublesome issue that was present throughout the entire development process was the lack of space and resources on the new build servers, which caused our continuous integration pipeline to fail constantly, sometimes forcing us to retry builds up to 24 times. This slowed down the development process in general, as we could not to merge any pull requests before the build passed.

6.2.2.5 GDPR and security requirements

During the research phase, we talked to our client about security and data storage as student information was likely to be stored with the use of our tool. As the tool is presumed to run on the servers of the University we would need to know what security measures our tool should adhere to for a secure operating environment. However, even after communicating with the Data Steward and the Head of ICT of the EEMCS faculty, we could not get a specific list of security requirements. We resolved this issue by implementing security to the best of our abilities, for the areas we identified to be critical for security. See section 3.19 for more implementation details.

6.2.2.6 Tools with poorly written/documented code

While searching for tools we often encountered tools that contained little to no documentation. For example, the only tool that provided a schema of the output that the tool produced was PyLint. This reduced the ease with which we could include more metrics. We could not do very much about this, other than running the tools on a variety of inputs and deducing a schema ourselves.

Many tools are not kept up-to-date with languages' newer features. For example, many are unable to parse Java's local variable declarations using the 'var' keyword. The only way to fix this is to fork the tools or submit merge requests ourselves.

Other tools had very poor code quality (e.g. Test Smells) and often crashed when ran. Some tools like this, that contained functionality that we needed were forked, major issues were removed and then included in our project as a local dependency.

6.2.2.7 Project size

As the project grew larger, we experienced that it became increasingly difficult to understand how the entire system works. There will always be parts of the tool that you haven't touched in a while and someone else has changed. This requires clear communication and documentation to make sure that everyone can continue to work on every aspect of the tool without having the need for the original developer to explain how it works. To this end, we used GitLab's wiki feature to document new or newly changed parts of the tool.

6.2.3 SIG

Over the course of our project, we had to submit our code to SIG twice. We then would receive feedback from them as to where we could improve our code. The feedback consisted of a score between 1 and 5 and an explanation of what could be improved. AuTA scored a 4 in the first code review. They explained their score by noting that the AuTA codebase had several long methods (unit length) and a number of methods with a lot of parameters (unit interfacing). SIG provided three examples of methods that suffer from these problems for both. After receiving these hints we have looked at the methods pointed out by SIG, as well as other methods that have a similar size or number of parameters.

The methods pointed out by SIG that stood out in terms of unit interfacing did have many parameters. However, with our use of dependency injection, reducing the number of parameters by using setters would have made the codebase less maintainable and more difficult to test. The unit size metric was more helpful and we indeed re-sized a number of methods we found were unnecessarily large. Some of these long methods were kept as the effort required did not weigh up to the additional maintainability gained by refactoring them. We decided against treating the metrics used by SIG and keeping some of these methods how they were. The second round of feedback did not arrive on time to be included in this report.

6.2.4 Testing strategy

In all components of the project, extensive tests were used to test all components of the project. We used JUnit5 to unit test the Java components (core, worker, and J2V8), Jest to test the JavaScript components AuTA.js, and unittest to test Python (Trampoline). Continuous integration was used on all repositories to test each component.

The user interface, AuTA-UI, was tested manually. We chose not to use any automation frameworks (e.g. Selenium) to test this, as the constant changes that were being applied would have forced us to rewrite the automation stages constantly. This would have taken a lot of time that we simply did not have. Manual testing allowed us to quickly make changes that would otherwise have required large changes in the tests. In addition, it is entirely possible that AuTA's user interface will be replaced by a more generic Labrador user interface, that will be used to manage the Queue, AuTA and the chatbot (Queue&A). These will all be part of Labrador, see section 2.5.1. However, if the user interface is to be used long-term, automated tests would be useful to verify that the UI behaves as expected.

Furthermore, we used an end-to-end testing strategy using the AuTA Gitlab Auto DevOps integration, which is described in section 3.7. In the AuTA GitLab repository, two Trampoline jobs were started that zipped the repository and sent it to a server where an up-to-date version of AuTA was permanently running. This zip file was then analyzed by AuTA and a report was sent back to the GitLab repository. A similar strategy was used in Trampoline's CI to test Python metrics.

These jobs were used to check the validity of the feedback and test all components of AuTA and Trampoline. This proved to be very useful, as many bugs were found using these end-to-end tests in both AuTA and Trampoline. Areas of improvement were also identified in both projects, such as the need for upload suppressions in Trampoline when the .git folder became too large (the nginx reverse proxy for the core server refused files larger than 128 MB).

Although the overall testing strategy was sufficient for the project, there are possible areas of improvement. For example, unit tests with mocks were used to test the Controllers - classes that handle API calls - in the core. It is considered bad test design to instantiate Spring's Service layer in a unit test, which made it practically impossible to test method and class security. This had to be tested manually using the UI. This could be improved by using Spring Integration Testing to instantiate both the Controller and Service levels. We chose not to do this, as manual testing of the UI also tested most of these features. To improve the maintainability of the project, integration tests that are more fine-grained than the full end-to-end tests would be required.

6.2.5 Code quality

The continuous integration pipeline was based on the GitLab Auto DevOps pipeline, which performs a large set of analyses [63]. The default configuration, which includes code quality, SAST, dependency scanning, container scanning, DAST, and deploying, was too generic for our application. Since these stages often failed, were excessively slow, or generated useless feedback, we disabled many of them and supplied our own stages for checkstyle and SpotBugs to remain sure of a high level of code quality.

Annotations were also used to enhance Java's type system and safety to document which parameters, fields, and variables were allowed to be null and which were not, which prevented many potential bugs [64]. Other annotations were used to enhance the documentation and aid the IntelliJ IDE in generating feedback, including method contract specifications and immutability markers [65].

6.3 Evaluation

If we were to do another project we would try to improve on the following points:

- **MoSCoW** - When we started the project we create a list of requirements using the MoSCoW methodology. This became a very large list of requirements that we had to complete. Afterward, it turned out that the list of requirements was a bit large for a single project. In a future project, we should take a closer look at what would be a feasible amount of work to complete in a given amount of time.
- **Documentation** - Even before we started with the project we already made clear to each other that we should start with the documentation immediately after the project had started as to not have to write everything at the end of the project. Although we made a good effort throughout the project of keeping documentation up to date, we still got behind on documentation. In a future project, we should focus even more on keeping documentation updated throughout the project.

Chapter 7

Ethics

In this chapter, we will discuss the various ethical questions that might arise with the implementation of our tool.

7.1 Human vs. machine

A machine lacks empathy. Where a machine will mark something as wrong because some script said so, a human will be able to understand the context within which the code was written. They can use their own experience and can read comments or communicate with the student in other ways to decide whether failing the submission is fair. This is especially important when you consider that for many assignments the goal for the student is to understand the subject matter, and not necessarily to submit perfect code.

Additionally, the verdicts given by machines can be wrong in some cases. It is not unthinkable that AuTA might encounter a bug when processing a student submission, leading to an incorrect result. Depending on the circumstances this could mean that a student is given a passing verdict where the submitted code was not sufficient, or their submission is marked as insufficient when it should not be.

It is for these reasons that we implore that there will always remain a human in the loop and that students will always have a way to appeal the decision of the tool. AuTA should never have the last word with regards to the verdict and the grade; it is merely a means to an and, a way to reduce the workload for the course instructors and TAs.

7.2 Personal information and data retention

In order for the tool to provide its service, it needs to store analysis results together with some way to distinguish between students. With sufficient information, it becomes possible to retrieve the information submitted by an individual or a group. This, on itself, is not an ethical issue: students agree by enrolling that the university is allowed to retain their data for a certain amount of time.

The problem lies in the fact that AuTA does not assign a limited lifetime to data, for all it knows the information will remain in its database forever. While this data may be useful for student-student benchmarking, it should certainly be anonymized after a number of years. Currently, the system has no support for this.

Chapter 8

Discussion

In this chapter, we will discuss the product that we have delivered and the process in general. We begin by comparing the requirements that we created at the start of the project to the final product.

8.1 Requirements

During the research phase, we created a list of requirements using the MoSCoW method. Almost all requirements in the 'must-haves' list (see appendix A) were implemented. The following requirements were not:

- **Provide support for Python** - Python is supported, but not as extensively as Java. For example, test file detection does not work properly. The first course that AuTA might be used for are Java courses. This is why we chose to focus on Python to a lesser degree.
- **Modification (and recalculation) of metrics after initial submission** - For now, AuTA is used primarily to produce feedback for students. If the assignment's metrics change students can submit their code again to get new feedback. If grading is added to AuTA as a feature, the ability to recalculate the results for all submissions will become more important.
- **No access to identifiable information** - Students never have access to identifiable information. However, admins, instructors, and TAs do. These users have access to all assignments, which means they can also see all submissions and the identity associated with that submission. This can be fixed relatively easily, by adding course-level access control.
- **Simple interface to configure Dynamic Analysis** - The only way to configure dynamic analysis (the Docker runner) is by providing a Dockerfile. We did not have enough time to simplify this interface.
- **Weblab integration** - This depends on the developers of WebLab. Our API documentation has been updated, and will be sent to the developers which will allow them to integrate AuTA with WebLab.

A limited amount of requirements from the other categories have been implemented. We focused on the requirements that would allow AuTA to be used for specific introductory courses at the start of next year. The requirements were created for a more general applicability of the tool, which is why the list is so extensive.

8.2 Security

The user-facing access control needs to be improved. For example, no instructors or TAs should be able to access any data associated with a course they have not been assigned to.

As mentioned in section 3.8, encryption for JUMP will also need to be improved to prevent a man-in-the-middle attack from intercepting submission jobs. This was not a priority during the project, as this part of the communication will happen behind the TU Delft firewall. However, we still think this needs to be fixed before releasing the tool.

8.3 Feedback quality

For each course, the type of feedback that is required will be different. The first programming assignments of the Object Oriented Programming course is less about code quality than programming to meet a strict specification explaining what the solution should be able to do. In these cases, using Docker to perform specification tests will be more useful than static analysis results. This feature is almost ready, but not quite there yet.

At the time of writing this report, AuTA will be more useful for courses where code quality starts to become more important and assignments are less about programming to satisfy a specification. For example, Software Quality and Testing and the Context Project courses. The reports that AuTA generates for students are very useful to quickly identify potentially problematic areas of the code; the feedback provided may not be useful to students in all cases.

AuTA generates feedback for students by comparing the metrics calculated for the submission to values that the course instructor or TA can configure manually. They can choose to add tips for students indicating how they could improve, and why the warning or failure was produced. See appendix J for an example of a report that is provided to the student.

Metrics can be useful to estimate code quality. However, there are always exceptional cases where from a software engineering perspective, this solution may be a good one despite exceeding some metric thresholds. In other cases trying to refactor code to treat a single metric may not be a good idea [66]. The feedback that we produce does not account for these cases and will therefore not always be useful. This is also the main reason that we decided to include a way for students to be able to suppress warnings or failures for a certain file – with proper justification.

Although improvements can be made with regard to the feedback our tool produces, we think our tool is ready to be implemented in programming courses.

8.4 Generating grades

One of our client's requirements was the ability to generate a grade based on the instructor's input. We were however not able to implement a simple and easy way to do this. The only way to currently do this is by use of a Docker container. This takes a significant amount of time to set up and we were not able to create an easier way for this due to the time constraints and the large backlog of features that needed to be incorporated into the tool. There should be another, simpler way to grade assignments based on the needs of the assignment that the instructor creates.

Chapter 9

Future work and recommendations

In this chapter, we will discuss what we think the future of the tool would be and what additions might still be very useful that aren't available in the tool at this moment.

9.1 System score generation

The way AuTA currently calculates the system score is without the test coverage that SIG uses to calculate their system score. This is because AuTA currently doesn't support test coverage as a metric. SIG has solved this by using a form of static test coverage estimation as described in [47]. They build a call graph and based on that they check which methods are being called. We believe that this is not a very solid way of calculating metrics as a class with 10 complexity and one test method is probably not that well tested but counts the same as a simple getter test. We, therefore, think we can best try to retrieve this from existing data (such as the coverage metric on GitLab) or by running the unit tests using the Docker integration. For other submissions, we will have to try and use the less accurate version that SIG uses to still return a system score that included the test coverage.

9.2 More languages

To expand the reach of the tool, more languages can be added. Currently, AuTA has support for Java, Python and AMD64 AT&T-style assembly. C and C++ have very basic support like line length and cyclomatic complexity. Scala support, for example, could be added to allow more courses to use the tool. These languages will have the most added benefit as these are the languages that are most used in the Computer Science Bachelor after Java and Python.

9.3 Easy Docker

During analysis, a Docker container can be created which takes the students solution and checks it for specific tests that the creator of the Docker file wants to check for (which includes the solution's unit tests). We think that improvements can be made here: it would be nice to have an easier way to specify the programs that should run in a container instead of providing a Dockerfile and there is also no way to format and use the results of the programs run within the container for display and further analysis, respectively.

9.4 Annotation suppressions

Per-submission parameters are specified using a file (.auta.yml) that can be added to the project. The primary usage of this file is to specify files that should be excluded from the analysis. This is useful for when you have files of which you are aware that they do not pass the criteria set out by the tool but that you want to keep that way anyway and have a good justification for. It would be better if there was a way to suppress a specific warning for a specific part of the code instead of the whole file. This would be possible by allowing the suppressions to target specific metrics or by using annotations in a similar way as for example SpotBugs uses them. We would advise that also in these annotations a justification is required so that it will be easy to understand why a part of the checks is suppressed.

9.5 Entity tree

As mentioned in section 3.11, when a submission arrives in the worker the entity is built down to file entities. This means that other entities, such as fields, methods, and classes are missing from this tree. These are added by the analyzers and after analysis entities with the similar names are merged. We rely on Lizard to add entities with correct names. However, Lizard does not parse all code correctly (e.g. default methods in interfaces).

We could reduce the number of errors made when building or merging the entity tree during analysis by building the entity tree beforehand using a dedicated parser for that language. An example of a tool we can use to do this is ANTLR4 [67]. Although there is no official ANTLR4 grammar for Java 11, we have already created a subset of our own to parse method names in tool results.

During or after analysis, we could then easily match names created by analyzers to the correct names, created by the parser. This would also help to make our tool less dependent on Lizard. If the names cannot be matched, we can use a best-guessing algorithm to try and match the names anyway.

9.6 Machine Learning

Machine learning methods may offer some way to improve parts of our tool. Machine learning is especially useful for classification, which is done in multiple places in our tool. We have identified a number of things where we think machine learning may improve the methods we currently use.

- **Test code detection** - There are many different edge cases that do not work with our current method for test code detection. For example, test helper files and projects that do not use JUnit.

- **Method name guessing** - As mentioned in section 9.5, even if we build the entity tree before analysis there will still be analyzers that produce entity names for which no matching entity can be found in the tree. A machine learning algorithm could be trained to help handle these edge cases.
- **Language detection** - The method we currently use to do language detection, which is described in section 3.17 is not ideal. GitHub Linguist is made for git repositories specifically and many of the submissions that we will receive will not be part of a git repository.

9.7 Support for huge projects

In this section, we describe what needs to be done in order to let AuTA work with very large projects (>30000 ELOC). Projects that are larger than this threshold are slow to generate reports for. In addition, results for these projects are too large for MongoDB which by default has a document size limit of 16 MB.

9.7.1 Report generation

Analysis report generation is currently handled by the core. As we described in section 5.2 this creates a bottleneck in the core. We also described the possible solution using workers for report generation, thus reducing the load on the core and possibly decreasing analysis time. With the structure that is already in place, workers could easily be extended to generate their own reports. This is mostly necessary for larger projects, such as those that are produced during the context project.

9.7.2 MongoDB

In order to reduce the size of the submissions, we should remove the results from the submission container. Results could then be stored on disk using a simple file store, instead of in MongoDB. MongoDB never queries the results and results can still be uniquely coupled to a submission using the submission ID. The only service that requires the results object (entity tree) after the analysis is the report generator.

9.8 Multiple Languages

Some projects, especially larger ones, consist of code written in multiple languages. Currently, AuTA only supports a single language per assignment, on top of language auto-detection. This means, that adding both Python and Java metrics in a single project is impossible. Support for more than one metrics could prove very useful for these kinds of mixed projects. Currently, the language auto-detection tool used, GitHub Linguist, already supports detecting different languages in code, which could be useful in implementing multi-language support in the future.

9.9 Grading

Currently, the only way that a specific grade, besides our single maintainability grade, can be returned is through a Docker container. This is, however, a cumbersome task to set up. Therefore there should be an easier way to return a grade for an assignment or part of an assignment. This can be achieved by adding another field to the metric scripts that state how many points a student can get for that particular metric or how many they will lose per mistake. Care must, however, be taken not to return a grade in the range of 0-10 as students may associate this with an actual grade that they received for the assignment while it only serves as an indication towards the student, TA, and instructor.

9.9.1 Fraud detection

When a grading system is implemented we can look at spotting potential fraud. When a grade suddenly spikes we can suspect that fraud has been committed. AuTA should detect these sudden spikes and alert a TA or instructor to it. They can then take a closer look at it to make sure that the flagged submission was indeed fraudulent.

9.10 Future of AuTA as part of Labrador

As progress on Labrador continues, AuTA 2.0 will be one of Labrador's micro-services. It will be updated along with the creation of Labrador and will be part of the eventual release of the Labrador project. This means that that depending on how Labrador is eventually going to function, AuTA might need some minor changes to function properly as a Labrador micro-service. Other tools that AuTA works with, CPM for instance, might be changed as well. Altogether we can not say for sure what will have to be changed, because Labrador is still in the early stages of development as of writing. Keeping these possible changes in mind remains important in the future development of AuTA, however.

Chapter 10

Conclusion

In this chapter, we give a summary of what we have created and how we expect it to be used by the TU Delft.

Over the course of 10 weeks, we have created AuTA, a tool that will become part of a larger education tool suite called Labrador. In this context, AuTA will be used for automated code analysis of student programming assignments.

AuTA was created to reduce the workload on teaching assistants, who spend a lot of time reviewing student submissions. By performing both static and dynamic analysis, the tool is able to provide teaching assistants with an overview of where student code does not meet certain quality thresholds. For example, if the code is too complex or does not meet certain styling standards. The tool supports multiple languages, including Java and Python. Using the user interface, teaching assistants or instructors are able to select which metrics they want to be calculated and what the thresholds for these metrics are.

The tool also provides feedback directly to students, by generating a report that highlights where their code does not meet quality thresholds. In addition, students are provided with tips that explain why exceeding these thresholds should be avoided, and how they could improve their code.

AuTA has a modular architecture, with components that perform different tasks split over packages and repositories. Workers, which are responsible for the actual code analysis, are shipped as a Docker image and can be managed by Kubernetes. Workers are then created and destroyed automatically depending on their workload.

This makes AuTA scalable and ensures high throughput. In practice, this means that even if all students submit their assignments at the same time, they will have to wait no longer than a few minutes to receive their reports.

AuTA has built-in integrations for GitLab and CPM. The well-documented REST API and bindings such as auta.js will allow for further integration with many other platforms used at the TU Delft, such as WebLab.

Although Labrador will not be ready at the start of next year, in its current state AuTA will be able to be used as a stand-alone tool during various courses. Nevertheless, some improvements can still be made, such as those described in Chapter 9.

Acknowledgements

We would like to thank Otto Visser for the opportunity to continue work on our context project as our final project. We would also like to thank our coach Annibale Panichella who guided us through the project. Furthermore, we would like to thank everyone who participated in our interviews to get a complete overview of the requirements. Finally, we would also like to thank everyone who provided their old homework submissions for us to test with.

Appendices

Appendix A

MoSCoW

##	Priority	Requirement	Completed
43	Must	Add assignments via the API	YES
44	Must	Add submissions via the API	YES
45	Must	Have a documented API	YES
58	Must	Provide a means to authenticate users	YES
03	Must	Provide support for Python	YES
14	Must	Modification (and recalculation) of metrics after initial submission	NO
46	Must	Provide support for Java	YES
47	Must	Easily add new quantitative and qualitative code measures	YES
48	Must	Provide support for Assembly	YES
57	Must	Measure code complexity, general style, lines of codes	YES
16	Must	Upload large numbers of files for checking (>800).	YES
18	Must	Provide buildpack for GitLab Auto DevOps	YES
49	Must	Provide API endpoint for CPM	YES
60	Must	Document the entire architecture and working of the tool	YES
20	Must	Provide an overview of the results per student	YES
21	Must	Give detailed line-by-line feedback via GitLab	YES
22	Must	Give detailed line-by-line feedback via WebLab	limited
50	Must	Give detailed line-by-line feedback via CPM	YES
25	Must	No access to identifiable information	limited
26	Must	Encrypted communication	YES
28	Must	Audit all third party tools for external communication	YES
33	Must	Download reports	YES
35	Must	Add users	YES
36	Must	Assign users as assignment owners	YES
37	Must	Simple interface to configure Dynamic Analysis	NO
51	Must	Set thresholds for different metrics	YES
52	Must	Allow users to select which code measures should be measured	YES
53	Must	Add jobs	YES
54	Must	View running status	YES
55	Must	View previous jobs	YES
56	Must	Be accessible via the API and the front-end	YES
04	Must	Automatically remove idle workers depending on workload	YES
05	Must	Automatically start workers depending on workload	YES
38	Must	View worker status	YES
39	Must	Add workers on remote devices	YES
40	Must	Terminate workers	YES
41	Must	Add multithreaded worker awareness	YES
42	Must	Restart workers	YES

TABLE A.1: Must-have requirements

##	Priority	Requirement	Completed
06	Should	API and integration with GitLab front-end (context ED5)	NO
62	Should	Metrics specific for test code	YES
17	Should	Upload test cases, using which the student's programs will be assessed	limited
19	Should	Outputs grade	NO
24	Should	Detailed feedback for smaller assignments (per-assignment linting rules?)	NO
61	Should	Feedback generation through comparison with a benchmark of solutions	NO
30	Should	Comparison with other submissions using a benchmark repository	NO
32	Should	Upload template solutions which the student's programs will be compared to	NO
34	Should	Allow for integration of Single Sign-On	limited
59	Should	Containerize workers for usage in GitLab's Auto DevOps	YES

TABLE A.2: Should-have requirements

##	Priority	Requirement	Completed
07	Could	API and integration with GitLab statistic extraction (context ED6)	NO
08	Could	API and integration with fraud checking tool	NO
01	Could	Provide support for C	limited
02	Could	Provide support for C++	limited
09	Could	Measure memory usage, CPU usage, File system usage and Network usage.	NO
10	Could	Measure variable name quality	NO
11	Could	Provide a student code ranking	NO
15	Could	Profanity checker	YES
27	Could	Encrypted storage	NO
29	Could	Automatic GitLab issue creation based on tool feedback using GitLab API	NO
31	Could	Ability to select which other assignments to compare to in benchmark repository	NO

TABLE A.3: Could-have requirements

##	Priority	Requirement	Completed
12	Won't	Provide support for Scala	NO
13	Won't	Git usage quality analysis	NO
23	Won't	Allow students to log in to the web service	NO

TABLE A.4: Won't-have requirements

Appendix B

Data management plan

Data Management Plan[i]

Name and contact details	*Luc Everse – l.everse@student.tudelft.nl* *Tim van der Horst – t.a.i.vanderhorst@student.tudelft.nl* *Erik Oudsen – e.m.oudsen@student.tudelft.nl* *Ewoud Ruighaver – e.ruighaver@student.tudelft.nl*
Name of project and group	*Client – Otto Visser – o.w.visser@tudelft.nl* *Coach – Annibale Panichella – a.panichella@tudelft.nl*
Description of your research	*Creating a tool (AuTA2.0) which is able to analyse the quality of student code, which is submitted for programming assignments. The tool will give feedback to students.*
Project duration	Start: *23-04-2019* End: *05-07-2019*

About this Data Management Plan

Date written	*26-04-2019*
Date last update	*26-04-2019*
Version	*1*

Changes in this version of the Data Management Plan

Component	**Progress / Execution** *Please describe shortly what progress you have made, any questions or issues you have encountered and want to discuss, etc.*
1. Data collection	………
2. Data storage and back-up	………
3. Data documentation	………
4. Data access, sharing and reuse	………
5. Data preservation and archiving	………

1. Data collection

Describing the data you will be creating/collecting

1.1 Will the project use existing or third party data ?

- ☐ No
- ☐ **Own / group previous research**
- ☐ Academic collaborators
- ☐ Commercial collaborators
- ☐ **Publicly available database / archive**
- ☐ Specialist commercial data provider
- ☐ Other (please specify)

Describe shortly provenance, type and format of this data. Are there any restrictions or requirements for use of third party data such as licensing conditions?

The group will be building upon a previously created piece of software (created by the same group, called AuTA). AuTA was created under an AGPL v3 licensing condition. AuTA makes use of the following open source tools, with the following licenses:

- *Lizard – MIT License*
- *qDOX – Apache License*
- *Cacodi – MIT License*
- *ANTLR – 3 clause BSD License*
- *PUML – GNU General License*

We will also be using programming assignments from all members of the group for testing purposes. These programming assignments were created in the first two years of the Computer Science Bachelors Programme at the TU Delft.

1.2 What type()s of data will you collect or create, in what file format(s)?[ii]

Note that not all formats are long-lived. For sustainable access you best use the formats recommended by data archives, see for examples: http://datacentrum.3tu.nl/fileadmin/editor_upload/File_formats/Preferred_formats.pdf or http://www.dans.knaw.nl/sites/default/files/file/EASY/DANS%20preferred%20formats%20UK%20DEF.pdf

We will collect metadata about student submissions, including the student number and results of programming assignment quality analysis. This data will be stored in a .json format in a MongoDB database.

1.3 How will you collect and/or create your data?

Please describe shortly. Name any relevant protocols and/or standard in your area of expertise.

If the group is granted permission to access code submissions belonging to other students from previous years, data mentioned in section 1.2 will be created by using the automated code quality analysis tool we are creating.

1.4 What tools, instruments, equipment, hardware or software will you use to capture, produce, collect or create the data?

Please give the names of the tools and state if they are already available. If not, state how you intend to acquire them. If applicable, describe whether you use a paper or electronic labjournal.

The original tool we will use is called AuTA, and was created by Luc Everse, Tim van der Horst, Erik Oudsen, Ewoud Ruighaver and Bailey Tjiong

2. Data storage and security

Ensuring that all research data are stored securely and backed up or copied regularly during your research

2.1 Where will you store your data?

Please describe how safe storage is guaranteed. Specify your method if your data is collected and / or transported in different locations / countries.

- ☐ On university departmental network storage (J:)
- ☐ **On university personal network storage (P:)**

	☐ In a Virtual Research Environment (Sharepoint) ☐ Physical storage (e.g. USB, external hard drive) ☐ Cloud service (e.g. SURFdrive) ☐ Other, namely: …
	The programming assignments belonging to the creators of the tool will be stored locally on the authors computers for testing purposes. *Programming assignments belonging to other students are currently stored on the university network storage.* *The code will be stored on a repository on GitLab https://gitlab.ewi.tudelft.nl/eip/code-measures. This server is hosted on the TU Delft's local server.*
2.2	**Will your data be backed up?** *Please specify shortly for each storage device frequency, location of backups and who is responsible. Describe how you can restore your data in the event of data loss and who is responsible.*
	The data on the TU Delft's servers is backed up.
2.3	**Are there any commercialisation, ethical or confidentiality restrictions about handling your data?** *Please specify shortly.*
	☐**Contractual obligations** ☐**Requirements by law : protection of personal data (e.g. privacy law) : specify in 4.1** ☐**Requirements by law : copyright, intellectual property : specify in 4.1** ☐Ethical restrictions (e.g. ethical review) : specify in 4.1 ☐ Commercial considerations (e.g. patentability) ☐ Formal security standards ☐ No requirements ☐ Other, namely: ………
	If the group is granted access to other student's submissions, this will be covered by an NDA. AuTA will be covered by an AGPL license, and intellectual property will belong to both the students and the TU Delft. *Student code submissions are regarded as personal data and must be handled accordingly.*
2.5	**What are the main risks to data security?** *Please list risks, e.g. accidental deletion, falling into the wrong hands.* *Please describe what would happen if the data get lost or become unusable.*
	Third parties gaining access to other student's submissions or the quality analysis results the tool generates for those submissions. *Interception of communication between different services of the tool. Student submissions are communicated between services in a .zip format.*
2.6	**What measures do you take to comply with the security requirements and to mitigate the risks?** *Describe how you can restore your data in the event of data loss and who is responsible.* *If applicable, please describe procedures to ensure personal data are handled confidentially and who is responsible.*
	☐ **Access restrictions** ☐ **Encryptions** ☐ Data processing ☐ **De-identification / Anonymisation** ☐ Regular back-ups ☐**Master copy stored on university network storage** ☐Master copy stored elsewhere ☐ Other, namely: …

	Communication between services should be encrypted. *Databases where results are stored should be encrypted or access should be restricted to the tool, instructors of courses and TAs only.* *De-identification – name, student numbers and net-ids are stripped from the submissions. Each assignment is assigned a unique id, and it is up to the submitter of data (such as CPM, Weblab, Gitlab) to couple this to a student.*

	3. Data documentation Documenting your data to help future users to understand and reuse it
3.5	**What supporting information / documentation will you create to enhance understanding of the data ?** *Please describe shortly how peers should be able to understand the data. Examples are a readme.txt, lab journals, a codebook, survey questions etc. Is there a standard for documentation in your field? Describe at what moment in your research process you will add the documentation necessary to make sure the data is understandable for peers.*
	A database model will be created and documented. A manual will be created for the entire tool regarding communication between different services, uploading of assignments, saving configurations, etc.

	4. Data access, sharing and reuse Managing access and security, sharing your data
4.1	**Are there any restrictions placed on sharing / reuse of some / all of your data?** *Please account for not sharing your data. Reasons may be ethical, commercial, security-related, protection of personal data rules, intellectual property, copyright,*
	Code for AuTA will be made open source. Student results will be shared with TAs and course instructors and the student. No other data will be shared.
4.4	**Who has authority to grant (additional) access to your data?** *Please describe shortly.*
	☐ Only you ☐ A colleague from the project, namely: … ☐ Supervisor ☐ Funder ☐ Collaborator / research partner organisation ☐ Other, namely: …
	Otto Visser
4.5	**How will you manage copyright and Intellectual Property Rights issues?** *Who owns the data? How will the data be licensed for reuse? Please describe shortly your choices and their consequences.*
	Unsure

[i] This template is based on the 3TU data management plan, the University of Bath data management plan and the Data Management Checklist of the University of Western Sydney.

Appendix C

Architectural diagrams

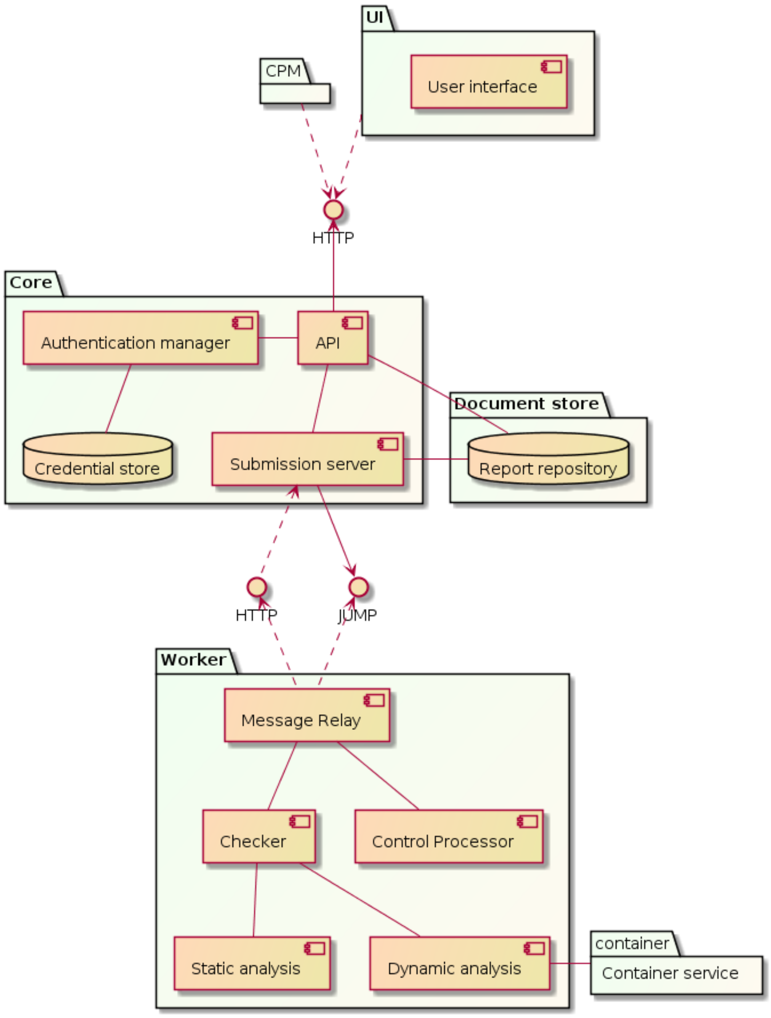


FIGURE C.1: System architecture as of AuTA 1.0. (Uses PlantUML syntax [33])

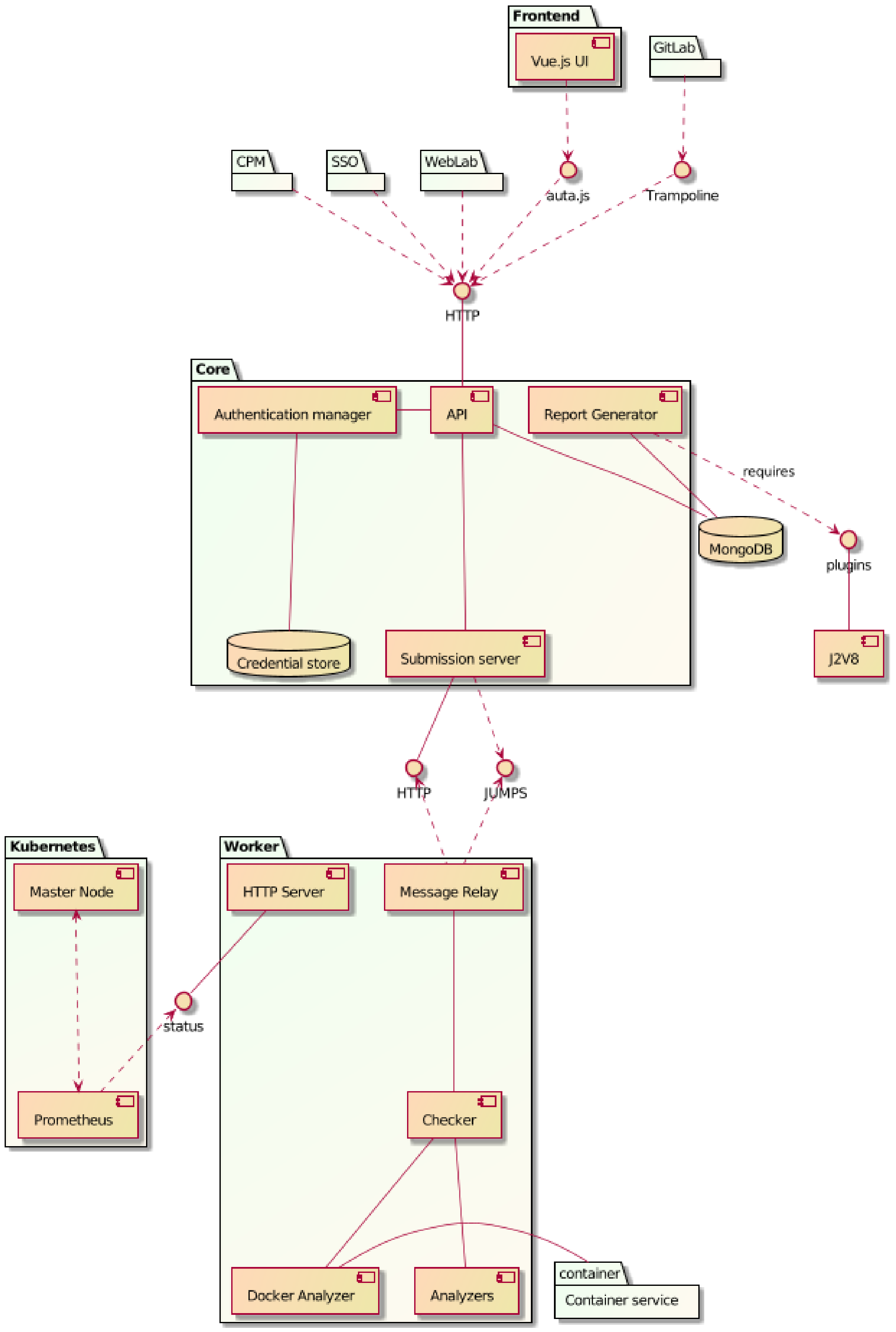


FIGURE C.2: System architecture in AuTA 2.0. (Uses PlantUML syntax [33])

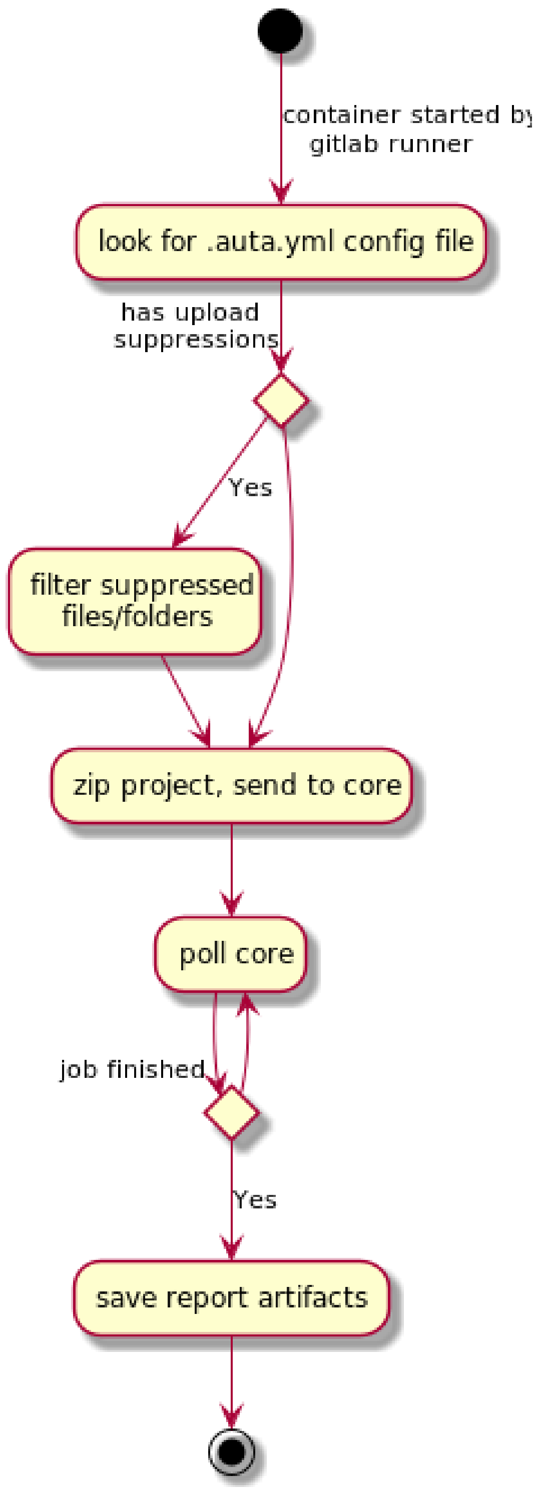


FIGURE C.3: Activity diagram for Trampoline. (Uses PlantUML syntax [33])

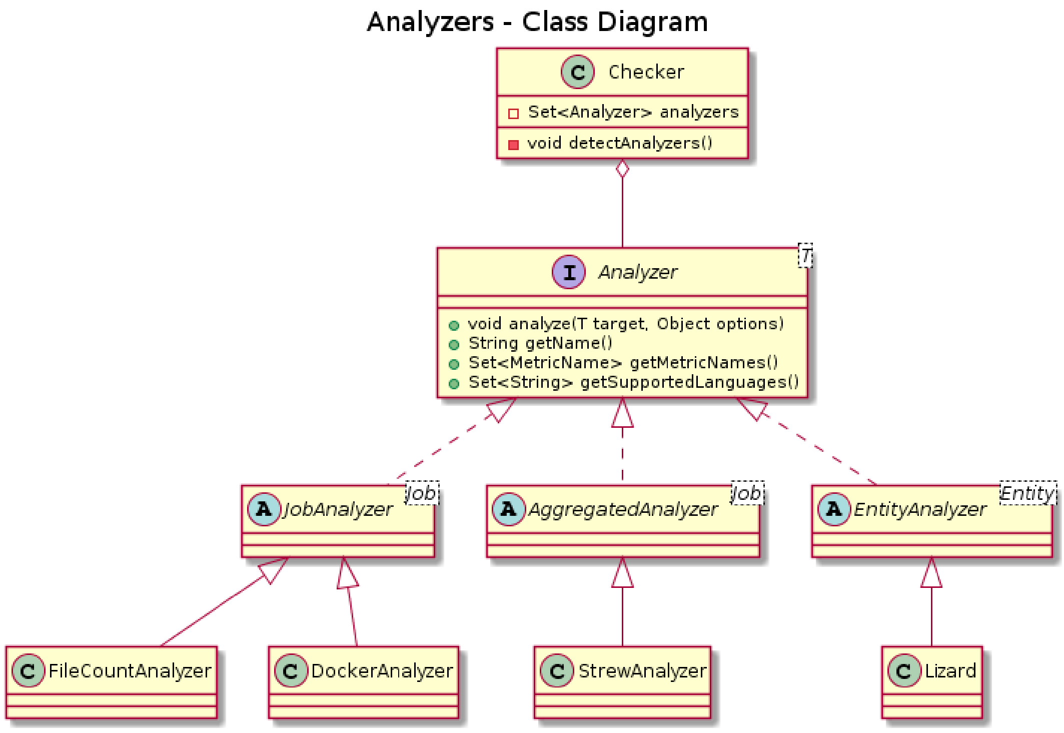


FIGURE C.4: Class diagram for analyzers and checkers in the worker. (Uses PlantUML syntax [33])

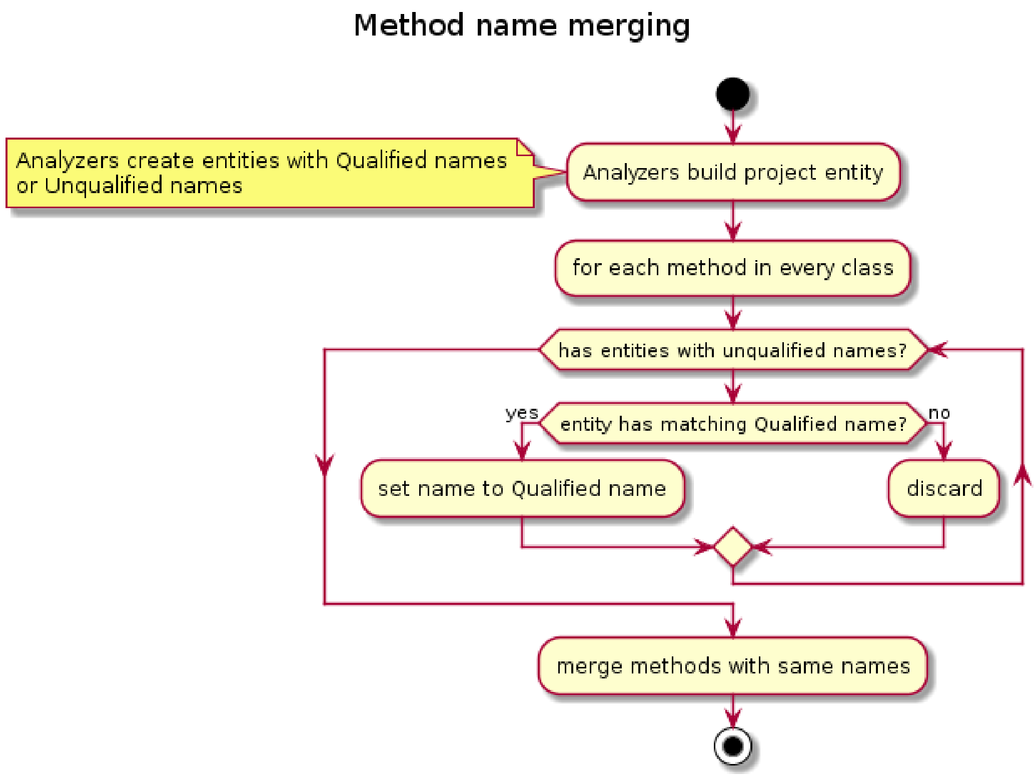

Method name merging
Analyzers create entities with Qualified names or Unqualified names
Analyzers build project entity
for each method in every class
has entities with unqualified names?
entity has matching Qualified name?
yes
no
set name to Qualified name
discard
merge methods with same names

FIGURE C.5: Activity diagram for the process of merging method names. (Uses PlantUML syntax [33])

Appendix D

Labrador overview

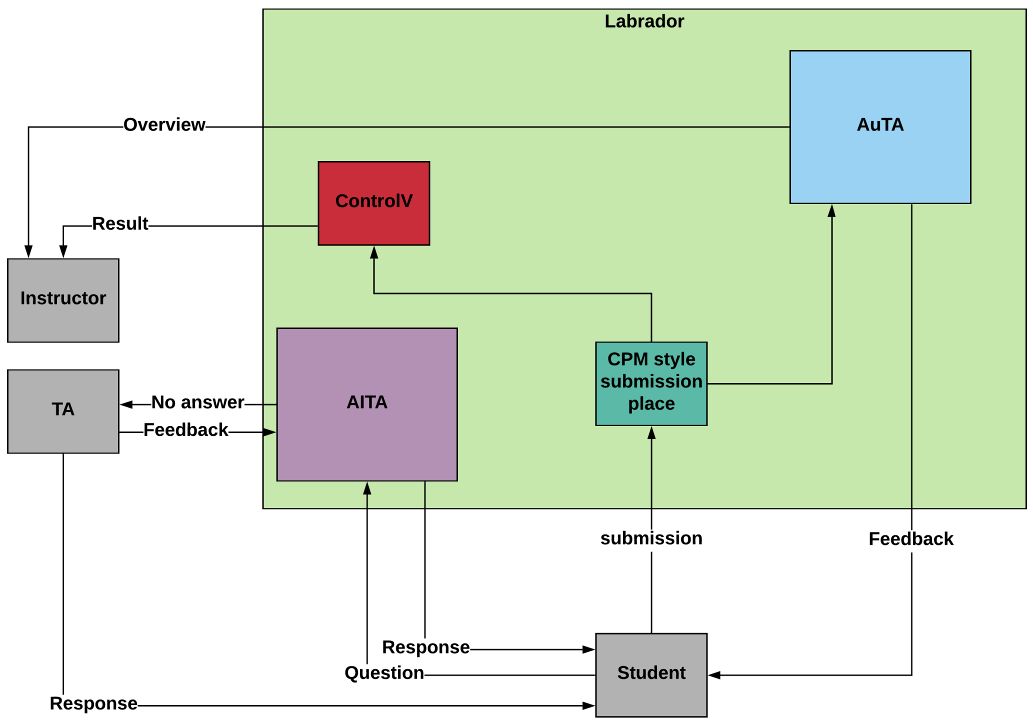


FIGURE D.1: Labrador simplified. (Not UML, just proposed overview drawings)

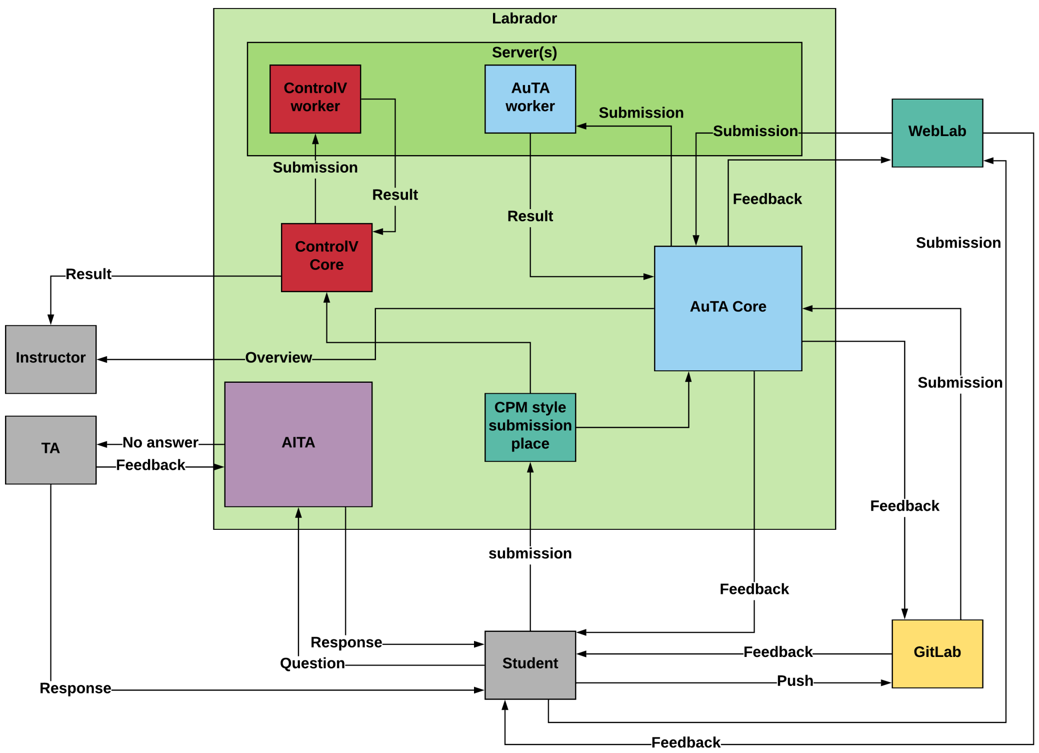


FIGURE D.2: Labrador detailed. (Not UML, just proposed overview drawings)

Appendix E

AuTA Submission flows

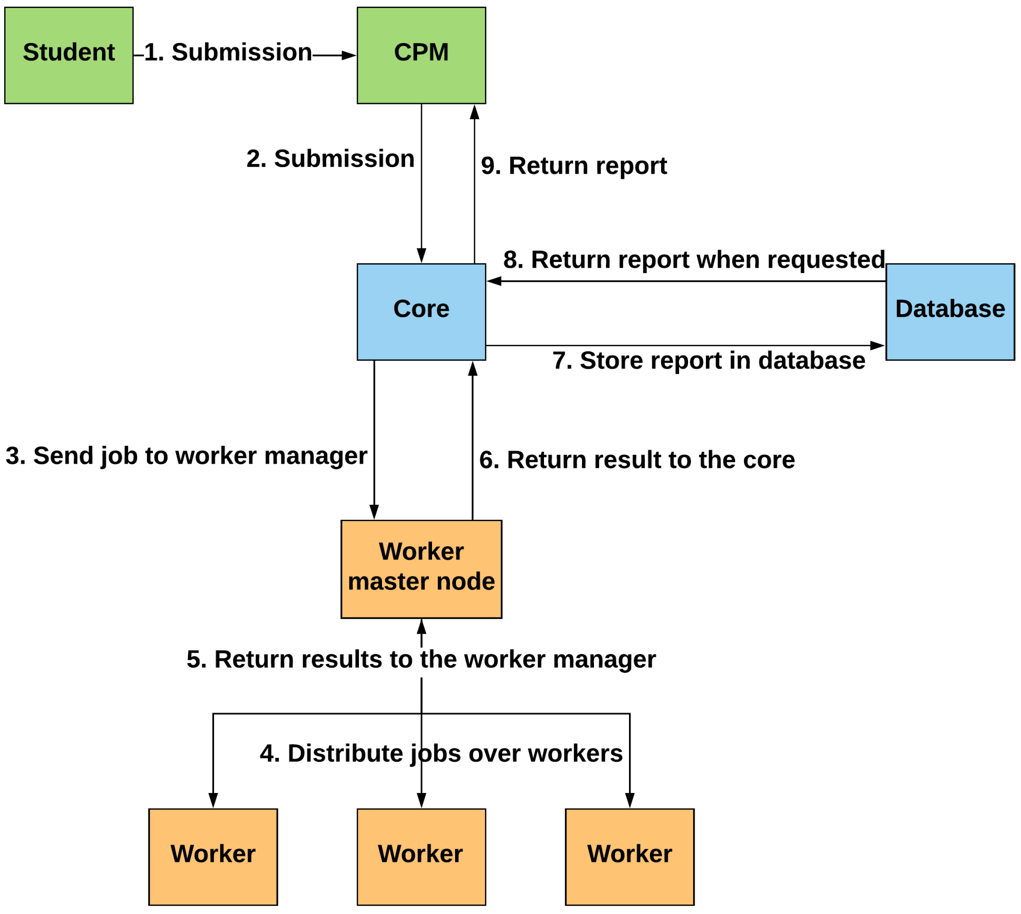

Student
1. Submission
CPM
2. Submission
9. Return report
8. Return report when requested
Core
Database
7. Store report in database
3. Send job to worker manager
6. Return result to the core
Worker master node
5. Return results to the worker manager
4. Distribute jobs over workers
Worker
Worker
Worker

FIGURE E.1: The flow of a submission through CPM. (Not UML, just proposed overview drawings)

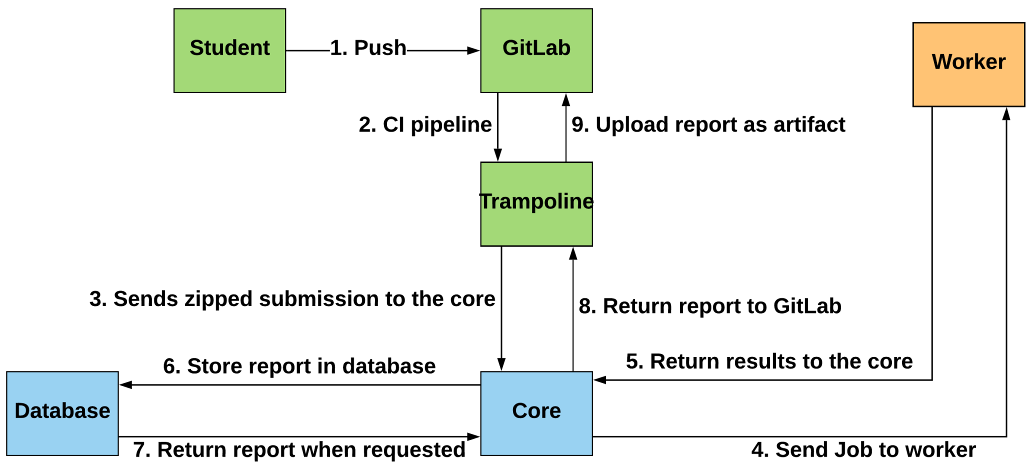

Student
1. Push
GitLab
Worker
2. CI pipeline
9. Upload report as artifact
Trampoline
3. Sends zipped submission to the core
8. Return report to GitLab
6. Store report in database
5. Return results to the core
Database
Core
7. Return report when requested
4. Send Job to worker

FIGURE E.2: The flow of a submission through GitLab. (Not UML, just proposed overview drawings)

Appendix F

Metric overview

1. Java
    - Abstract method count
    - Anonymous classes count
    - Assertions per test
    - Assignments count
    - Average assertions per testcase
    - Class effective lines of code
    - Comparison count
    - Constructor count
    - Coupling between objects
    - Cyclomatic complexity
    - Default field count
    - Default method count
    - Depth inheritance tree
    - Duplicate code blocks
    - Field count
    - field usage
    - File count
    - File effective lines of code
    - Final field count
    - Final method count
    - Javadoc exists
    - Javadoc violations
    - Lack of cohesion of methods
    - Lambdas count
    - Loop count

- Math operations count
- Max nested blocks
- Method count
- Method effective lines of code
- Number count
- Number of static invocations
- Parameter count
- Parenthesized expression count
- Percentage commented lines
- Private field count
- Private method count
- Protected field count
- Protected method count
- Public field count
- Public method count
- Relative coupling between objects
- Relative cyclomatic complexity
- Relative depth inheritance tree
- Relative number of assertions
- Relative number of testcases
- Relative weight method class
- Response for a class
- Return count
- Static field count
- Static method count
- String literal count
- Subclasses count
- Synchronized field count
- Synchronized method count
- Test method count
- Test production ratio
- Test smells
- Try catch count
- UML

- Unique words count
- Variables count
- Variables usage
- Weight method class

2. Python
    - Cyclomatic complexity
    - Duplicate code blocks
    - File count
    - File effective lines of code
    - Maintainability index
    - Method effective lines of code
    - PyLint
3. Assembly
    - Commented line count
    - File count
    - Recursion target

Appendix G

Manual

G.1 Cluster setup

We will first setup the cluster to enable auto scaling

G.1.1 Kubernetes setup

We assume `kubeadm` is already installed, but not initialized yet. Instructions on how to install `kubeadm` can be found here (https://kubernetes.io/docs/setup/independent/install-kubeadm/).
First thing to do is choose a subnet for the kubernetes pods. We suggest choosing one that is not already in use. In this example we will use the range `192.168.0.0/16`.

Running as a root user:

```
kubeadm init --pod-network-cidr=192.168.0.0/16 --ignore-preflight-errors= Swap
```

To allow `kubelet` to run without swap disabled, we should go into `/var/lib/kubelet/config.yaml` and change

```
failSwapOn: true
```

to false.
After which you should follow the steps described in the output of this command, it will also provide an example of a `kubeadm join` command, which should be executed on any machine that should be part of the cluster. Along with the step to disable failSwapOn. More on this can be found on the kubernetes docs (https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/).

Next, we will set up weave as our CNI plugin. This is simply done by executing the following command as any user on only the master node:

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=
"(kubectl version | base64 | tr -d '\n')"&env.IPALLOC_RANGE=192.168.0.0/16"
```

Where the `IPALLOC_RANGE` variable is the range we set earlier.

Now the cluster should be ready to start deployments, this can be tested with `kubectl get nodes`. Which should display all nodes in the cluster, each one should have the status `Ready`.

G.1.2 Prometheus

To allow K8s to use the Horizontal Pod Autoscaler on our custom metrics, we have to install Prometheus (https://github.com/coreos/kube-prometheus/). This will be a distilled version of the quickstart guide found on the Kube-prometheus GitHub repo.

```
git clone https://github.com/coreos/kube-prometheus.git
```

Now, we can move into the kube-prometheus directory and create the services on the cluster with the following command

```
kubectl create -f manifests/
```

Note: This command should be executed twice, don't ask me why but there's some weird race going on in K8s which won't create all the services in time.

Next, we will move into the /experimental/custom-metrics-api (https://github.com/coreos/kube-prometheus/tree/master/experimental/custom-metrics-api) directory and perform the `deploy.sh` script.

To be able to view the monitor externally, we will need to forward the ports of the pods to our master node

```
kubectl --namespace monitoring --address 0.0.0.0 port-forward svc/grafana 3000
&; kubectl --namespace monitoring --address 0.0.0.0 port-forward
svc/alertmanager-main 9093 &; kubectl --namespace monitoring --address 0.0.0.0
port-forward svc/prometheus-k8s 9090 &
```

For some reason K8s decided it was a good idea to have these commands run in the foreground, so the command above allows the port-forward to continue even as the shell is closed. It would probably be nice to run these as a service so it automatically forwards these ports even after a reboot.

G.1.3 AuTA worker

Next, we will create a worker YAML file. An example of this can be found here (https://gitlab.ewi.tudelft.nl/eip/code-measures/snippets/1992).

```
kubectl create -f worker.yaml
```

After this command, the cluster should be running with auto-scaling worker pods!

G.2 System setup

In this section we will explain how to run the core and a worker if no use is made of the setup explained above.

G.2.1 Runtime requirements

Core AuTA's core requires a Java 11 runtime and MongoDB running on localhost. Optionally Graphviz can be installed to enable UML generation. Another option is to add the J2V8 pluging (https://git.wukl.net/f00f/auta-j2v8-plugin) can be added for scipt execution outside the MongoDB database for better performance. Furthermore, the path `/srv/auta` must be readable and writable by the user AuTA runs as.

Worker Like the core, the worker requires a Java 11 runtime.

G.2.2 Usage

Core Jar:
Start the core as follow:

```
java -jar core.jar
```

Docker:

```
docker run f00f/auta-core
```

The server allows for set-up using username `admin` and password `admin`. Configure the server as needed and restart the process.

Worker Jar:

```
java -jar worker.jar name=$NAME host=$HOST api-token=$APITOKEN
api-protocol=$PROTOCOL api-port=$PORT
```

Argument	Description	Required	Default
name	The name of this worker	N	Generated
api-token	Worker api token	Y	N/A
host	The host on which to reach the core	N	localhost
api-protocol	Protocol to use for the API (http/https)	N	https

TABLE G.1: Arguments for the worker

Docker:

```
docker run -e {environment variables} f00f/auta-worker
```

As opposed to running the worker with the jar, all environment variables are mandatory.
A table of environment variables that should be passed to the container:

Environment variable	Description
COREADDRESS	Address to reach the core on
APIKEY	Worker API token
APIPORT	Port to reach the API on
APIPROTOCOL	Protocol to use for the API (http/https)

TABLE G.2: Arguments for the worker when starting from docker

G.3 Devops

AuTA can be integrated in GitLab's Auto DevOps using Trampoline (https://git.wukl.net/f00f/trampoline), which is a lightweight Docker container that can zip a repository, send them to AuTA's `/api/v1/assignment/aid/submission/sid/direct-upload` endpoint and accepts a `report.json` which can then be uploaded as an artifact. Optionally, AuTA can produce a more readable HTML formatted report if the optional `?format=html` parameter is passed by trampoline when polling the `GET /api/v1/assignment/aid/submission/sid` endpoint. This endpoint creates a new submission for the given sid, which should be equal to the SHA-1 commit hash that GitLab generates. If a submission already exists in AuTA's database (for example if the pipeline for a commit is run twice) then AuTA will delete previous submission data.

G.4 External Service Integration

G.4.1 GitLab

1. Create an assignment on AuTA. Make sure to select the correct language and metrics.
2. Add the following job to the .gitlab-ci.yml. Make sure to replace the assignment id with the id of the assignment you just created.

```
trampoline-feedback:
  image:
    name: "timvanderhorst/trampoline:auta"
    entrypoint: [""]
  variables:
    AUTA_ASSIGNMENT_ID: 5cda623347fc2b00068befa1
    AUTA_SERVER_URL: https://autatest.f00f.nl
    TRAMPOLINE_POLLING_TIME: 5
    AUTA_BENCHMARKING_METRICS: CYCLOMATIC_COMPLEXITY:METHOD,METHOD_EFFECTIVE_LOC:METHO
  stage: test
  allow_failure: true
  script:
    - cd /home/trampoline ; python3 run.py
  artifacts:
    paths:
      - report.html
      - benchmark_CYCLOMATIC_COMPLEXITY_METHOD.html
      - benchmark_METHOD_EFFECTIVE_LOC_METHOD.html
    expire_in: 1 week
```

3. Benchmarking metrics can be added by typing adding a new `METRIC_NAME:ENTITY_LEVEL` to the `AUTA_BENCHMARKING_METRICS` variable. Make sure to also add a new artifact, `benchmark_METRIC_NAME_ENTITY_LEVEL` to the artifacts paths list.

4. A new environment variable `AUTA_API_TOKEN` should be added to GitLab CI settings, where the value is the token provided by AuTA. Do not add this variable to the .gitlab-ci.yml as this token can be used to log in to AuTA.

G.4.2 CPM

1. Create an assignment on AuTA. Make sure to select the correct language and metrics.
2. Navigate to the workunit options page on CPM
3. in the `Script URL for automated checks` field, add the following URL:

   `https://autatest.f00f.nl/api/v1/cpm/[ASSIGNMENT ID]?auth-token=[API TOKEN]}`

   where [ASSIGNMENT ID] is replaced with the id of the assignment that was just created, and [API TOKEN] is replaced with the API token that can be used to access AuTA

Appendix H

Tool metrics

H.1 Test Smells Metrics

In this section, we give a list of test smells that the Test Smells Detector tool (see section 4.1) can produce.

- Assertion Roulette - This smell checks if there is more than one assertion per test. If there is, documentation must be present as to what each assertion does. If this is not the case, this test smell will be returned to the student. Because with multiple assertions it isn't clear what is the problem if the test fails.
- Conditional test logic - This smell checks if there is a conditional logic in the test. e.g. an if statement or a for loop. If this is present in the test method, this test smell will be returned to the student. Because the assertion is than dependent on the flow of the control blocks, this makes the assertion unpredictable and the test method less maintainable.
- Constructor initialization - This smell checks if there is a constructor for the test class instead of a `setUp()` method. If there is, this test smell will be returned to the student. Because, to initialize values in a test class a `setUp()` method should be used instead of a constructor.
- Duplicate assert - This smell checks if the same assertion is called more than once in a test method. This means that the type of assert as well as the parameters passed to the assert are equal to each other. If this is the case, this smell is returned to the student. Because, If the method needs to test the same condition using different values, a new method should be utilized.
- Eager - This smell checks if multiple methods of the production class are called in one test method. If this is the case, this test smell is returned to the student. Because if this test fails it is not clear which function of the production class is the problem.
- Empty - This smell checks if a test method has executable statements, if this is not the case this test smell is returned to the student. Because empty tests do count towards coverage and might give the developer the idea that his new production code is working as expected because the test will pass.
- Exception catching throwing - This smell checks if there is a try catch block in the test method. If this is the case, this test smell is returned to the student. Because the handling of exceptions should be done by JUnit's exception handler.

- General Fixture - This smell checks if everything that is assigned in the `setUp()` method is also used in the test method. If this is not the case this test smell is returned to the student. Because unnecessary work is done in the `setUp()` method.
- Ignored - This smell checks if a test is being ignored. If this is the case, this test smell is returned to the student. Because ignored tests only provide more overhead to compilation time.
- Lazy - This smell checks if methods call the same SUT method. If this is the case, this test smell is returned.
- Magic Number - This smell checks if there is a magic number present in the test method. If this is the case, this test smell is returned to the student. Because magic numbers don't show their meaning and are thus not maintainable.
- Mystery Guest - This smell checks if external resources are used in the test. If this is the case, this test smell is returned to the student. Because external resources result in unstable test methods and cause performance issues.
- Print Statement - This smell checks if a print statement is present in the test method. If this is the case, this test smell is returned to the student. Because they have no added value to the test and only serve a purpose for debugging.
- Redundant Assertion - This smell checks if an assertion will always be true or false. If this is the case, this test smell is returned to the student. Because they have no added value.
- Sensitive Equality - This smell checks if equality is checked after a `toString()` call. If this is the case, this test smell is returned to the student. Because of the possibility of the usage of `toString()`.
- Sleepy - This smell checks if a `Thread.sleep()` call is made in a test method. If this is the case, this test smell is returned to the student. Because this can lead to unexpected results as it might do something different on each machine.
- Unknown - This smell checks if there is an assertion in the test method. If there isn't, this test smell is returned to the student. Because a test without an assertion is not a test.

H.2 CK Metrics

In this section, we give a list of metrics that CK is able to calculate.

- Return count - Counts how many return statements there are in a given class or method. This can give an indication as to how complex the code is.
- Loop count - Counts the number of loop statements in a given class or method. e.g. for, while, do while, etc.
- Comparision count - Counts the number of times there was a comparison made i.e. == in a given class or method.
- Try catch count - Counts the number of try catch blocks in a given class or method.

- Parenthesized expression count - Counts the number of parenthesized expressions in a given class or method.
- String literal count - Counts the amount of time a String is instantiated in a given class or method.
- Number count - Counts the amount of numbers in a given class or method.
- Assignments count - Counts the amount of assignments that are made in a given class or method.
- Math operations - Counts the number of math operations that are performed in a given class or method.
- Variables count - Counts the number of variables that are in a given class or method.
- Maximum nested blocks - Gives the highest number of blocks nested together in a given class or method.
- Anonymous classes count - Counts how many anonymous classes are in a given class or method.
- Subclasses count - Counts how many subclasses there are in a given class or method.
- Lambdas count - Counts the amount of lambdas in a given class or method.
- Unique words count - Counts the amount of unique words in a given class or method.
- Method count - Counts the number of methods in a class, can also count private, public, static, final, synchronized, protected, abstract and default methods separately.
- Field count - Counts the number of fields in a class, can also count private, public, static, final, synchronized, protected, abstract and default fields separately.
- Lines of code - Counts the effective lines of code in a given method or class.
- Static invocations - Counts the number of static invocations in a given class or method.
- Response for a class - Counts the number of unique method invocations in a class.
- Lack of cohesion of methods - Calculates the lack of cohesion of methods in a given class.
- Coupling between objects - Counts the number of dependencies a class has.
- Weight method class - Counts the number of branch instructions in a class.
- Depth of inheritance tree - Counts the number of superclasses a class has.
- Variables usage - How much each variable was used in a method.
- Field usage - How much each field was used in a method.

Appendix I

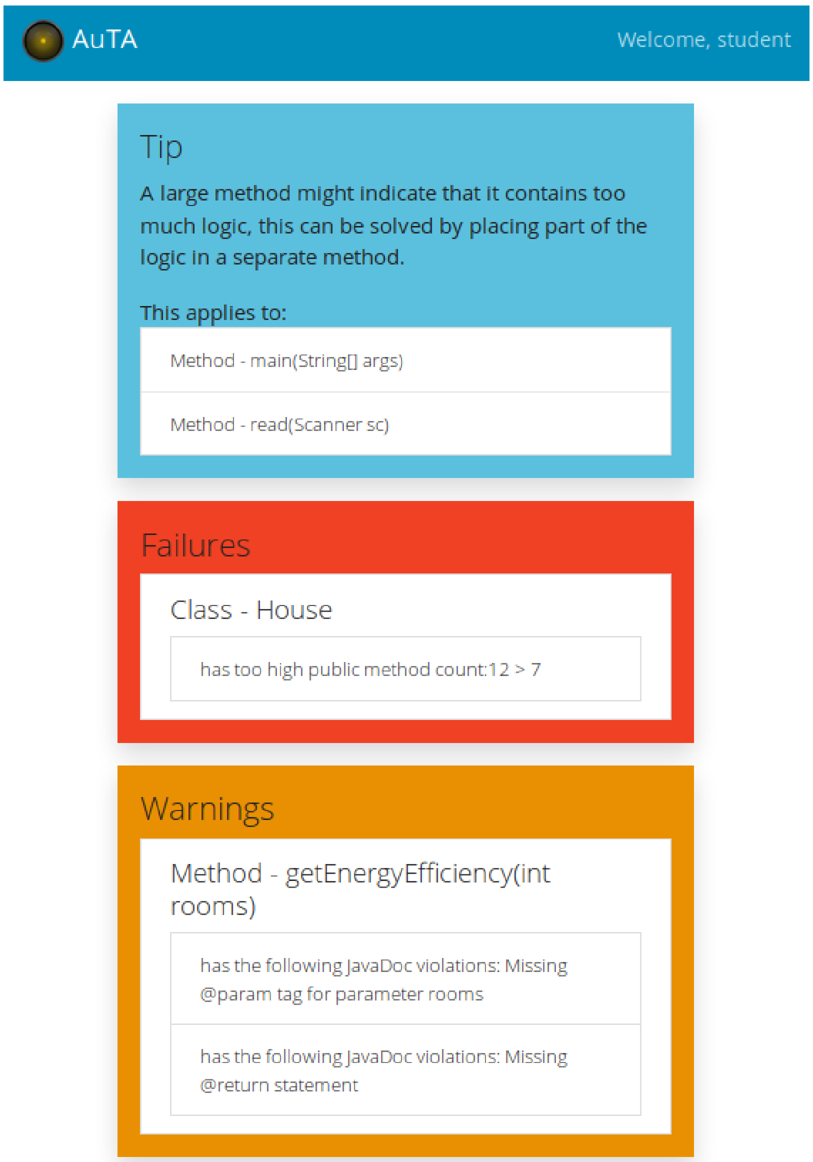
Project statistics

Repository	NLOC	Line coverage
code-measures	47891	76%
auta-ui	3606	N/A
auta-js	3214	98%
trampoline	698	97%
auta-j2v8	935	94%
Total	56344	

TABLE I.1: Lines of code and test coverage for each project repository

Appendix J

Example Reports



Tip

A large method might indicate that it contains too much logic, this can be solved by placing part of the logic in a separate method.

This applies to:

Method - main(String[] args)

Method - read(Scanner sc)

Failures

Class - House

has too high public method count:12 > 7

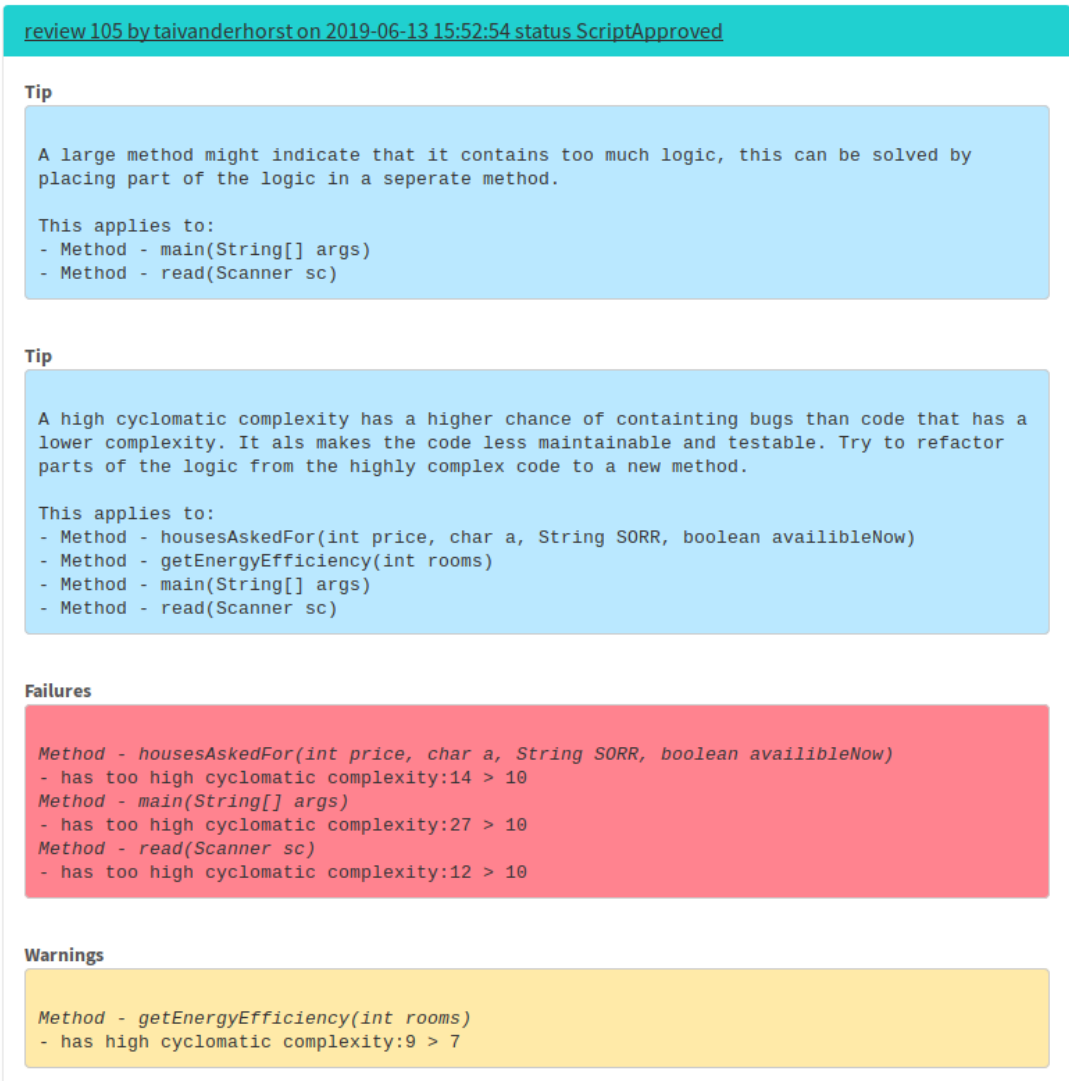
Warnings

Method - getEnergyEfficiency(int rooms)

has the following JavaDoc violations: Missing @param tag for parameter rooms

has the following JavaDoc violations: Missing @return statement

FIGURE J.1: An example of a report that is sent to the student directly (e.g. via GitLab)



review 105 by taivanderhorst on 2019-06-13 15:52:54 status ScriptApproved

Tip

```
A large method might indicate that it contains too much logic, this can be solved by
placing part of the logic in a seperate method.

This applies to:
- Method - main(String[] args)
- Method - read(Scanner sc)
```

Tip

```
A high cyclomatic complexity has a higher chance of containting bugs than code that has a
lower complexity. It als makes the code less maintainable and testable. Try to refactor
parts of the logic from the highly complex code to a new method.

This applies to:
- Method - housesAskedFor(int price, char a, String SORR, boolean availibleNow)
- Method - getEnergyEfficiency(int rooms)
- Method - main(String[] args)
- Method - read(Scanner sc)
```

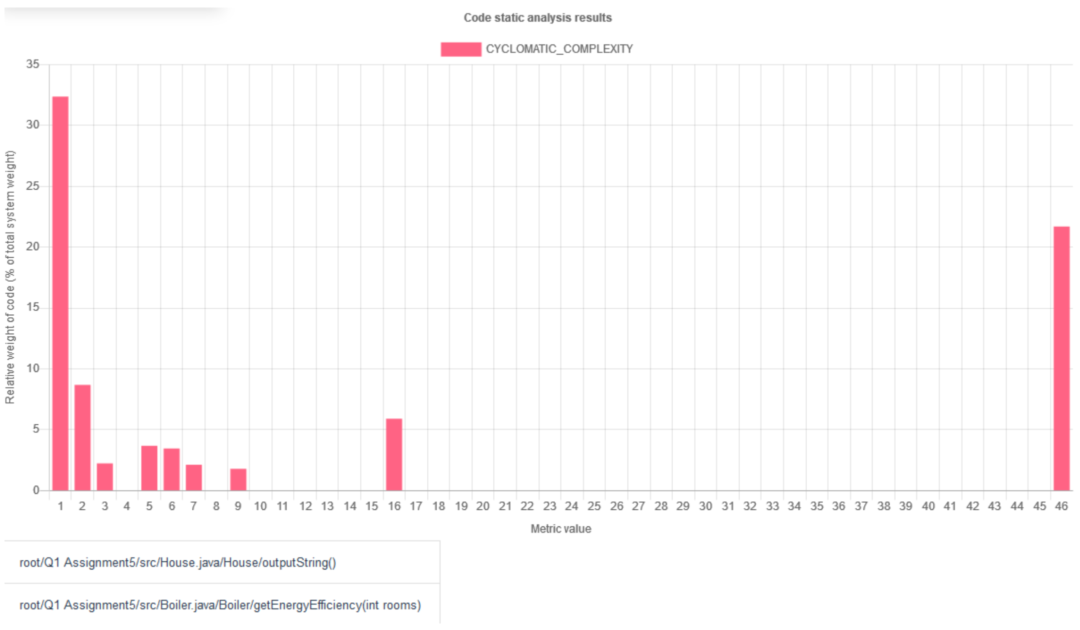
Failures

```
Method - housesAskedFor(int price, char a, String SORR, boolean availibleNow)
- has too high cyclomatic complexity:14 > 10
Method - main(String[] args)
- has too high cyclomatic complexity:27 > 10
Method - read(Scanner sc)
- has too high cyclomatic complexity:12 > 10
```

Warnings

```
Method - getEnergyEfficiency(int rooms)
- has high cyclomatic complexity:9 > 7
```

FIGURE J.2: An example of a report that is uploaded to CPM



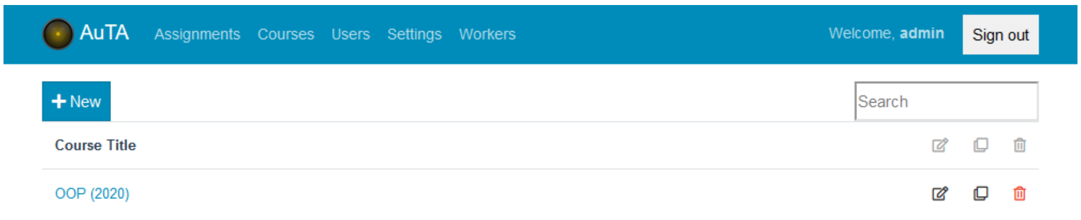
root/Q1 Assignment5/src/House.java/House/outputString()

root/Q1 Assignment5/src/Boiler.java/Boiler/getEnergyEfficiency(int rooms)

FIGURE J.3: An example histogram generated for cyclomatic complexity. Clicking on a histogram bar lists the entities with that cyclomatic complexity

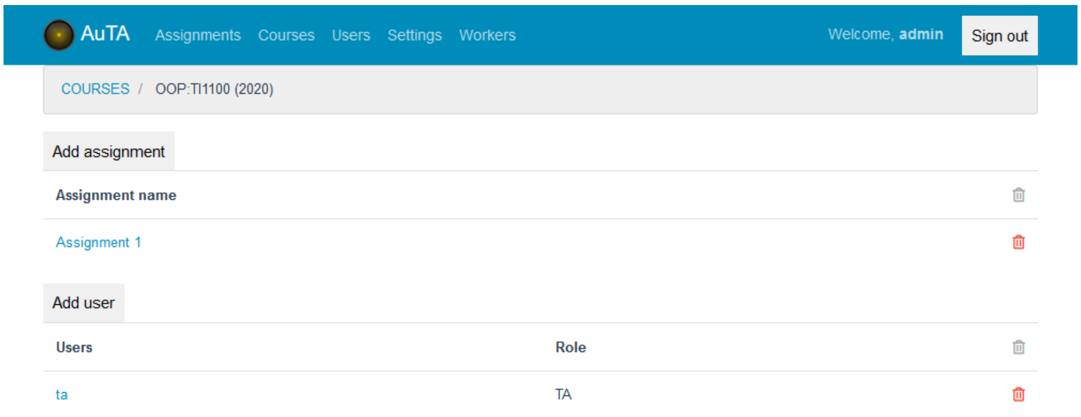
Appendix K

UI screenshots



Course Title
OOP (2020)

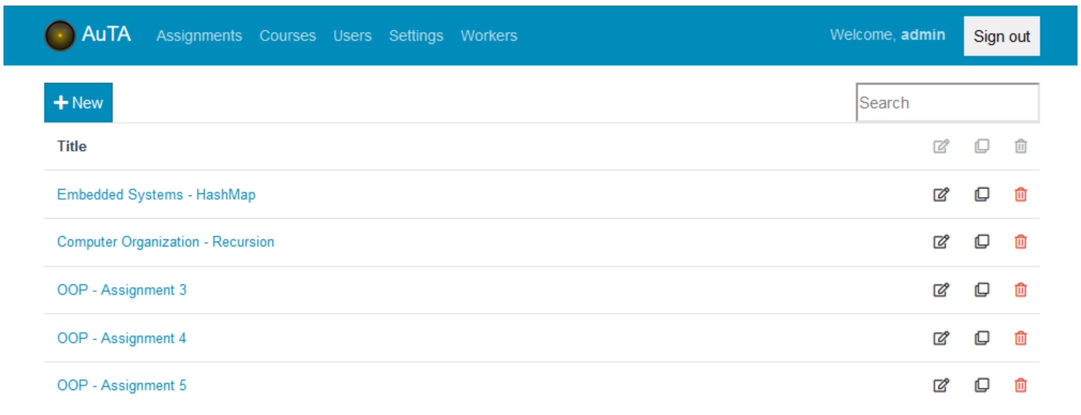
FIGURE K.1: The page that shows a list of all courses that have been created



Assignment name
Assignment 1

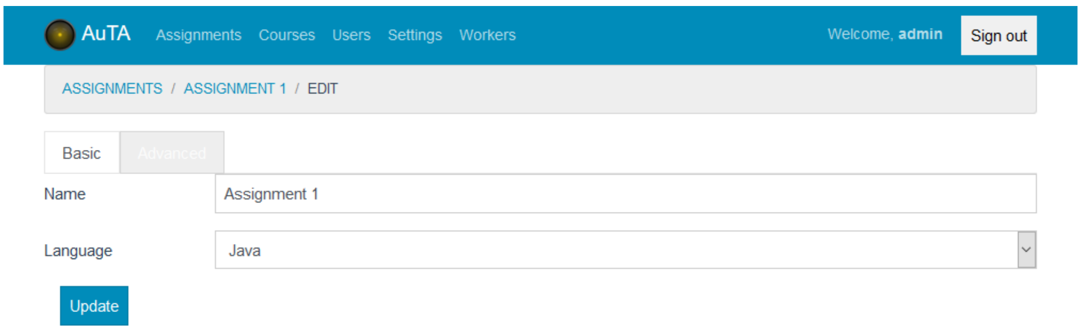
Users	Role
ta	TA

FIGURE K.2: The page for a specific course, where TAs or instructors can add assignments to that course, or add users to the course



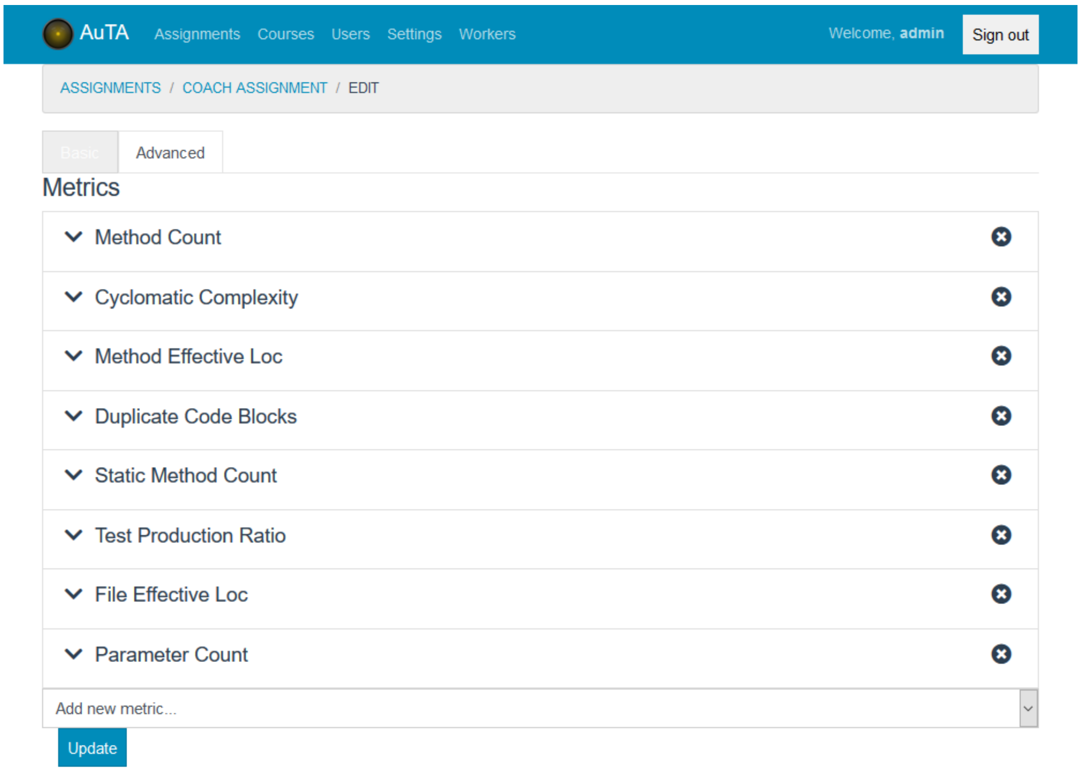
Title
Embedded Systems - HashMap
Computer Organization - Recursion
OOP - Assignment 3
OOP - Assignment 4
OOP - Assignment 5

FIGURE K.3: The page where all assignments are listed, with the option to edit or clone an assignment (which creates a new assignment with the exact same metrics and language)



Name	Assignment 1
Language	Java

FIGURE K.4: The assignment creation page, where the name of the assignment and language can be selected

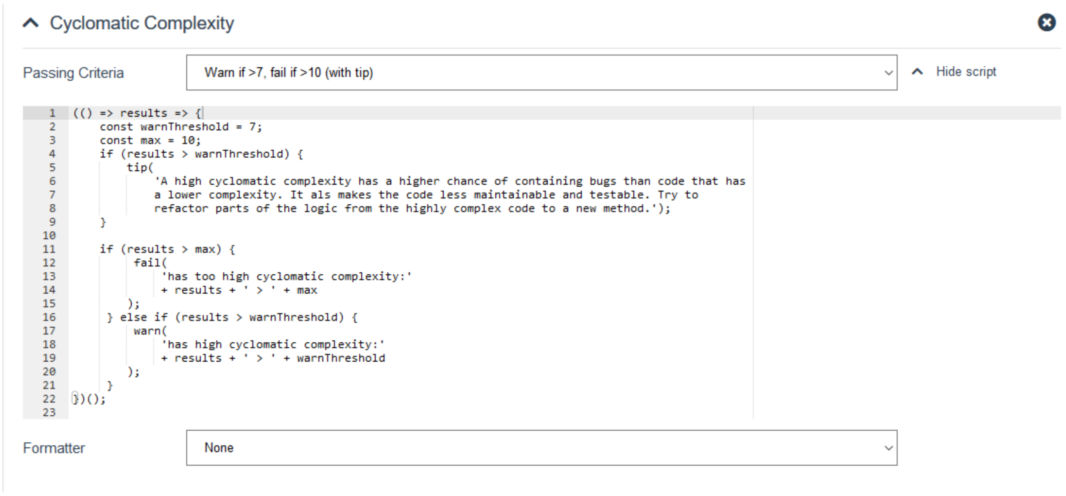


Metrics

- Method Count
- Cyclomatic Complexity
- Method Effective Loc
- Duplicate Code Blocks
- Static Method Count
- Test Production Ratio
- File Effective Loc
- Parameter Count

Add new metric...

FIGURE K.5: The page where metrics can be added to an assignment. The dropdown contains all metrics that are available for the assignment's language



Passing Criteria: Warn if >7, fail if >10 (with tip)

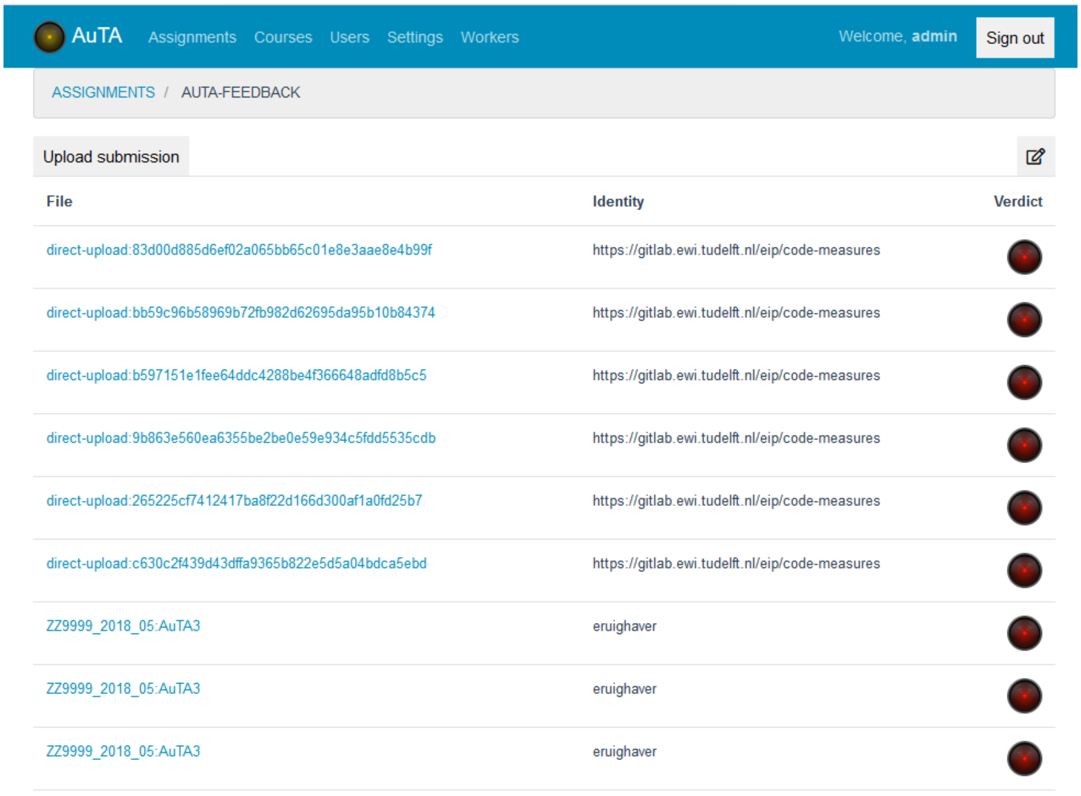
```
(() => results => {
    const warnThreshold = 7;
    const max = 10;
    if (results > warnThreshold) {
        tip(
            'A high cyclomatic complexity has a higher chance of containing bugs than code that has
            a lower complexity. It als makes the code less maintainable and testable. Try to
            refactor parts of the logic from the highly complex code to a new method.');
    }

    if (results > max) {
         fail(
             'has too high cyclomatic complexity:'
             + results + ' > ' + max
        );
     } else if (results > warnThreshold) {
         warn(
             'has high cyclomatic complexity:'
             + results + ' > ' + warnThreshold
        );
     }
})();

```

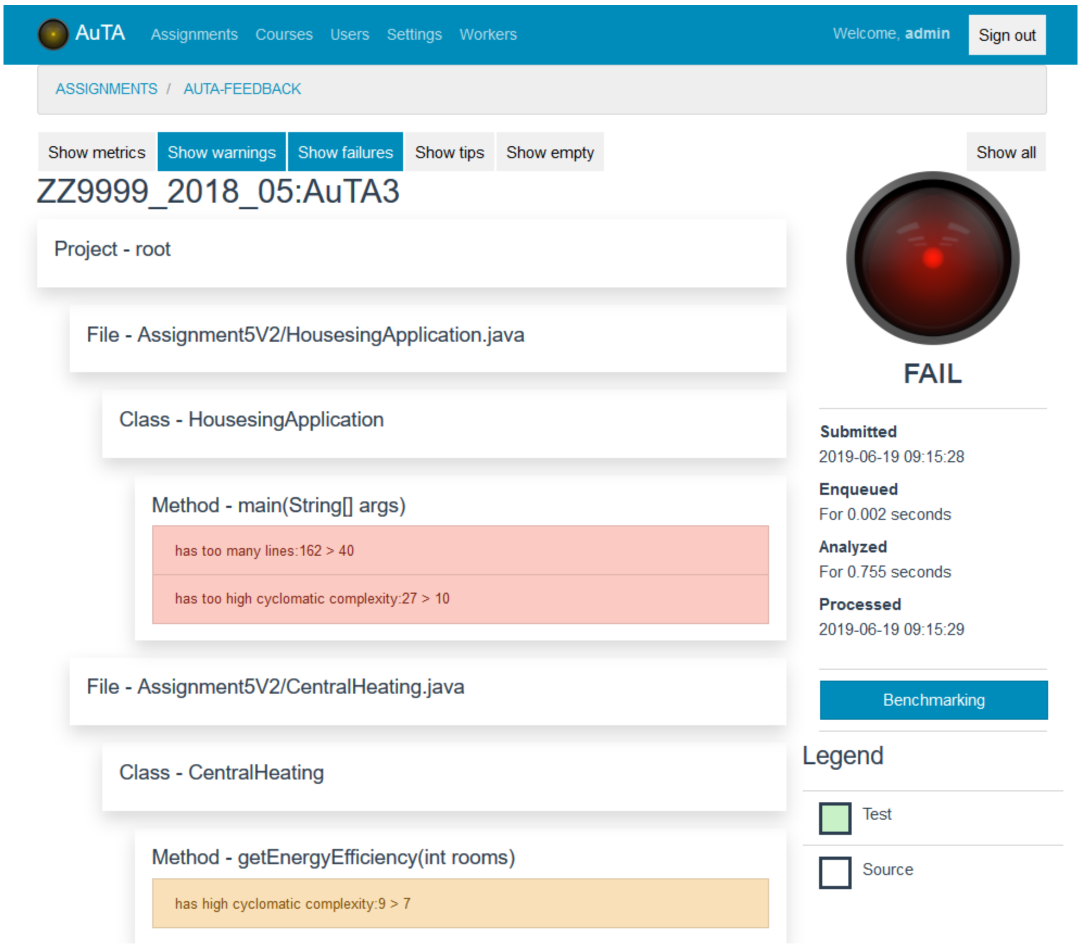
Formatter: None

FIGURE K.6: An example of a passing script that can be customized for each assignment



File	Identity	Verdict
direct-upload:83d00d885d6ef02a065bb65c01e8e3aae8e4b99f	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
direct-upload:bb59c96b58969b72fb982d62695da95b10b84374	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
direct-upload:b597151e1fee64ddc4288be4f366648adfd8b5c5	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
direct-upload:9b863e560ea6355be2be0e59e934c5fdd5535cdb	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
direct-upload:265225cf7412417ba8f22d166d300af1a0fd25b7	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
direct-upload:c630c2f439d43dffa9365b822e5d5a04bdca5ebd	https://gitlab.ewi.tudelft.nl/eip/code-measures	🔴
ZZ9999_2018_05:AuTA3	eruighaver	🔴
ZZ9999_2018_05:AuTA3	eruighaver	🔴
ZZ9999_2018_05:AuTA3	eruighaver	🔴

FIGURE K.7: The page that lists all submissions for a specific assignment. The image on the right side of each submission indicates the verdict associated with that submission - red is fail, blue is error, green is pass and yellow is pass with warnings



ZZ9999_2018_05:AuTA3

Project - root

File - Assignment5V2/HousesingApplication.java

Class - HousesingApplication

Method - main(String[] args)

has too many lines:162 > 40

has too high cyclomatic complexity:27 > 10

File - Assignment5V2/CentralHeating.java

Class - CentralHeating

Method - getEnergyEfficiency(int rooms)

has high cyclomatic complexity:9 > 7

FAIL

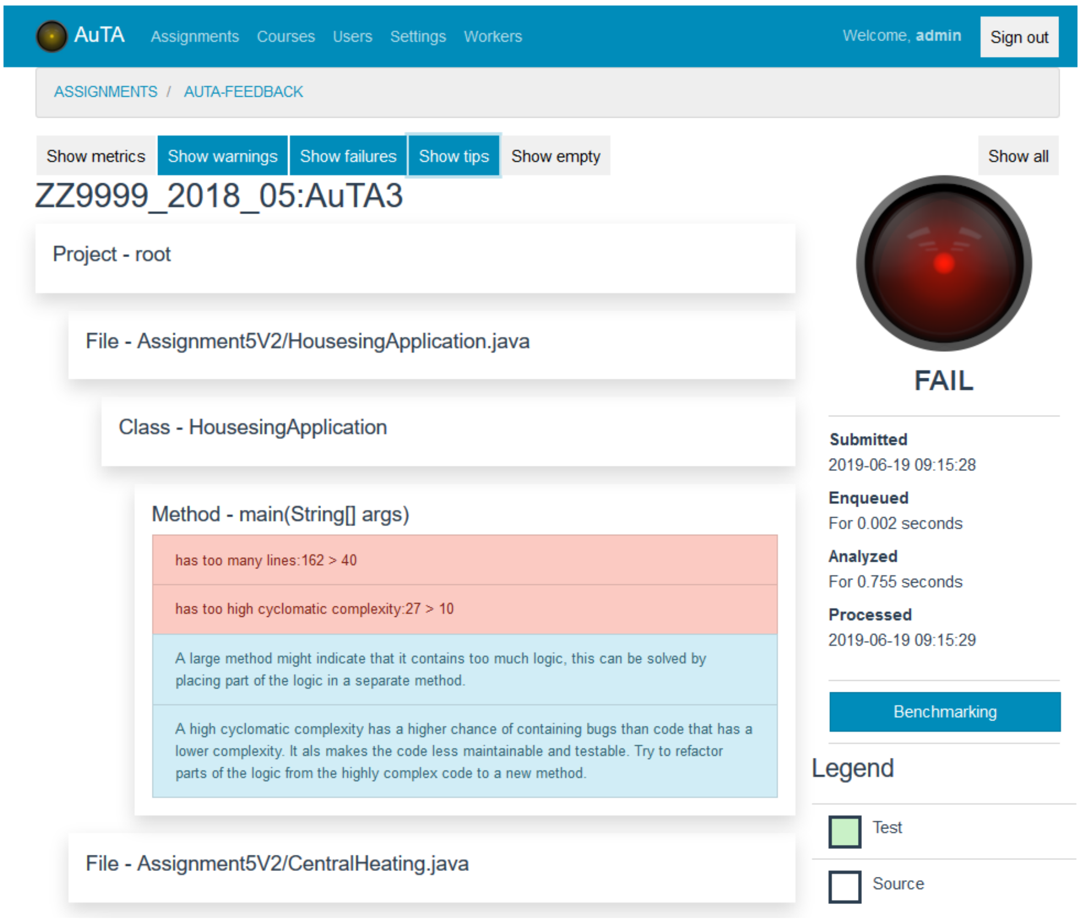
Submitted
2019-06-19 09:15:28

Enqueued
For 0.002 seconds

Analyzed
For 0.755 seconds

Processed
2019-06-19 09:15:29

FIGURE K.8: The page for a specific submission. Each entity with warnings or failures is listed. Users can select whether they want to show warnings, tips or failures (or all of them). The status bar on the right shows information about how long the analysis took for that assignment



ZZ9999_2018_05:AuTA3

Project - root

File - Assignment5V2/HousesingApplication.java

Class - HousesingApplication

Method - main(String[] args)

has too many lines:162 > 40

has too high cyclomatic complexity:27 > 10

A large method might indicate that it contains too much logic, this can be solved by placing part of the logic in a separate method.

A high cyclomatic complexity has a higher chance of containing bugs than code that has a lower complexity. It als makes the code less maintainable and testable. Try to refactor parts of the logic from the highly complex code to a new method.

File - Assignment5V2/CentralHeating.java

FAIL

Submitted
2019-06-19 09:15:28

Enqueued
For 0.002 seconds

Analyzed
For 0.755 seconds

Processed
2019-06-19 09:15:29

Benchmarking

Legend

🟩 Test

⬜ Source

FIGURE K.9: The page for a specific submission, with tips showing

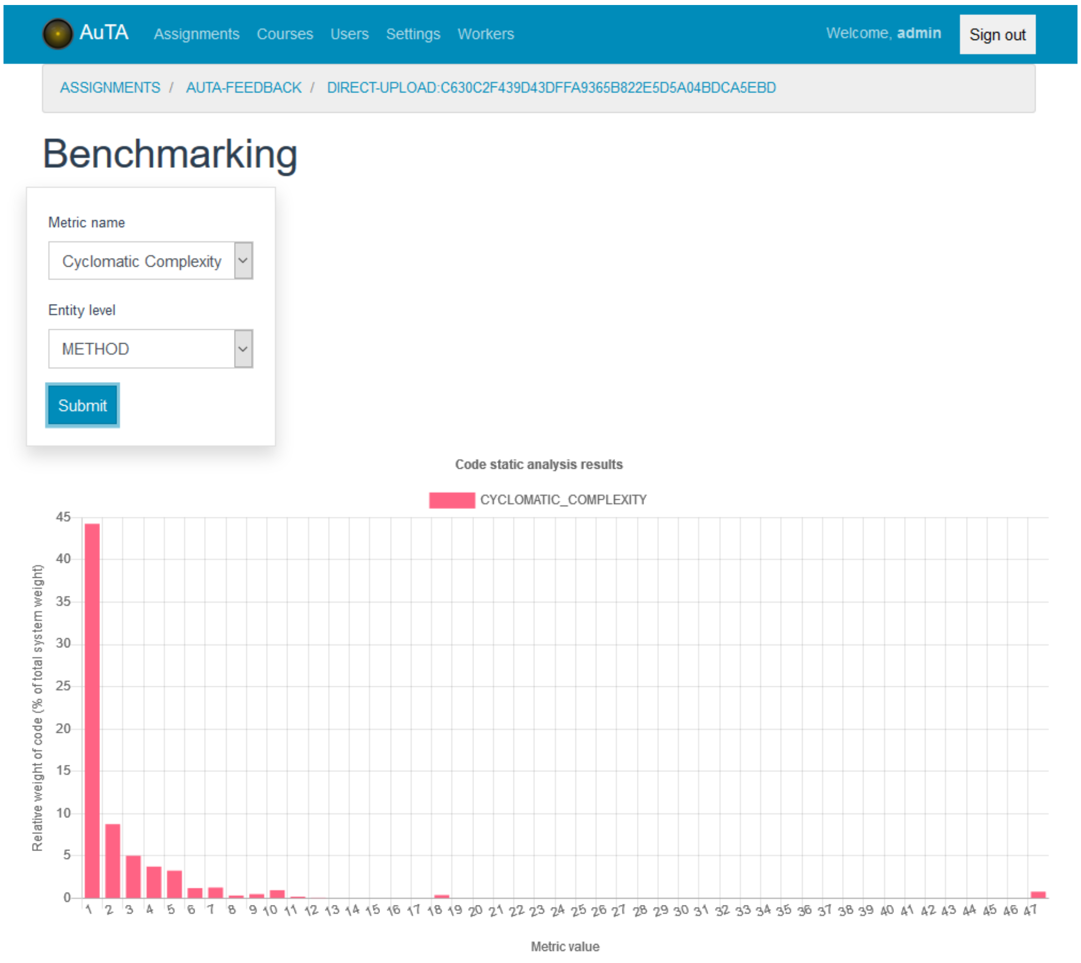
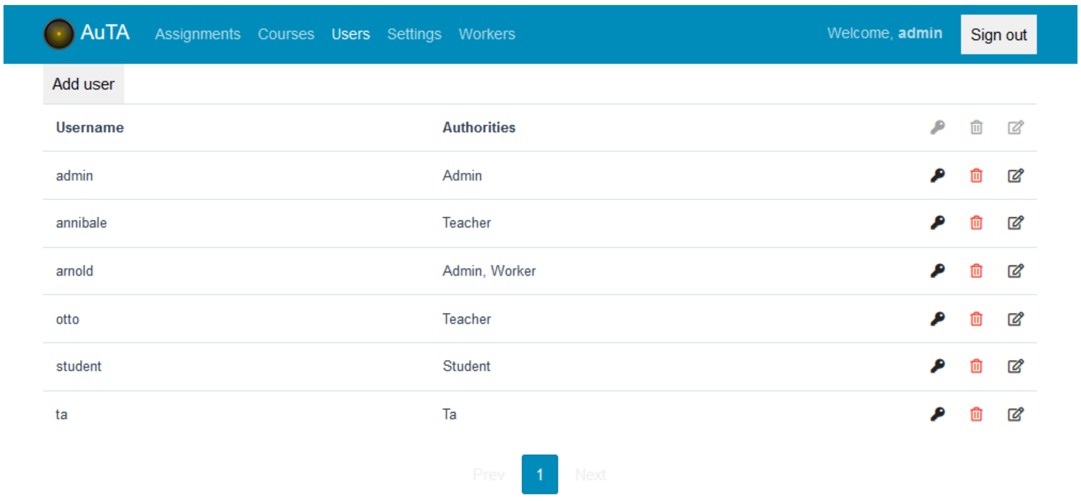
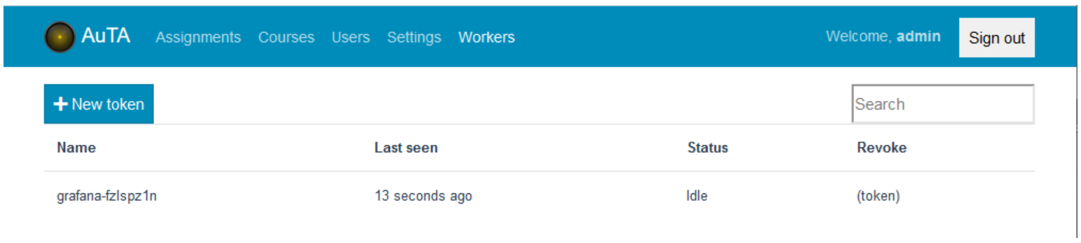


FIGURE K.10: The page where the user can generate histograms for each metric, where the bar height is equal to the total entity weight with that metric value. Entities are aggregated into bins with the same metric values



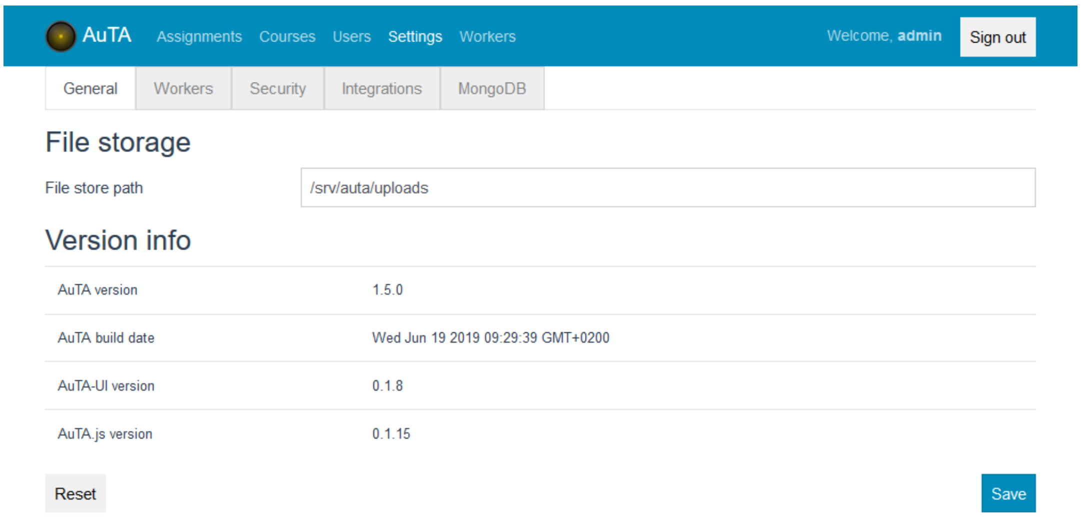
Username	Authorities
admin	Admin
annibale	Teacher
arnold	Admin, Worker
otto	Teacher
student	Student
ta	Ta

FIGURE K.11: The page that shows all users that have been registered in the database. On this page, users can edit their passwords. Admins can create new users, or edit their roles.



Name	Last seen	Status	Revoke
grafana-fzlspz1n	13 seconds ago	Idle	(token)

FIGURE K.12: The page that lists all workers that are currently connected to the system



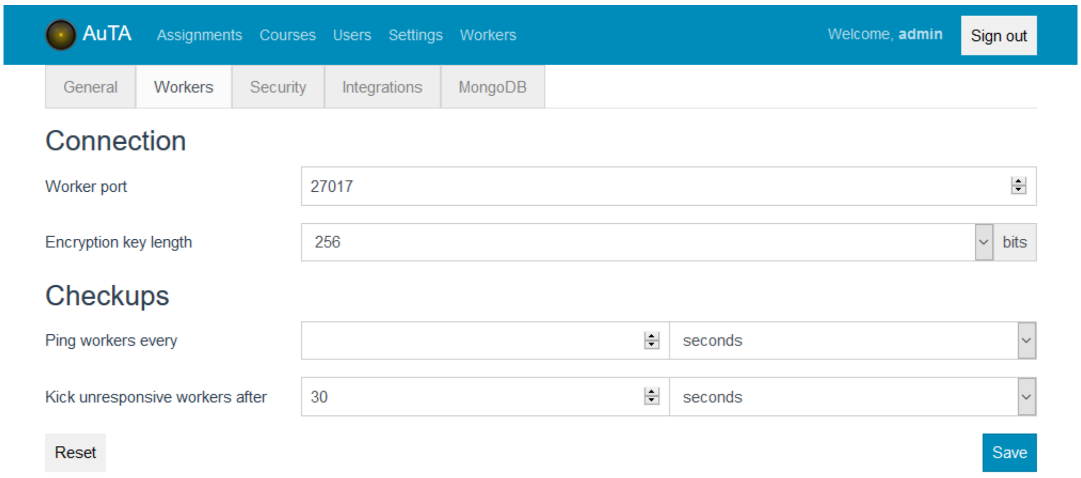
File storage

File store path: /srv/auta/uploads

Version info

AuTA version	1.5.0
AuTA build date	Wed Jun 19 2019 09:29:39 GMT+0200
AuTA-UI version	0.1.8
AuTA.js version	0.1.15

FIGURE K.13: The general settings page



Connection

Worker port: 27017

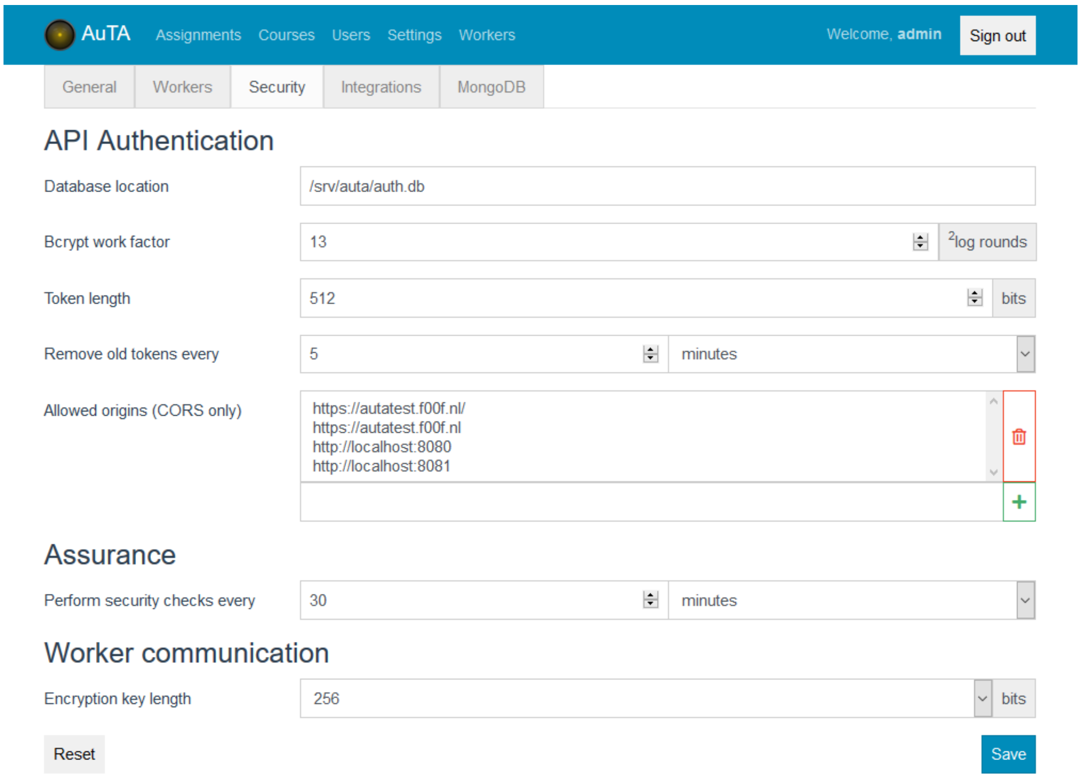
Encryption key length: 256 bits

Checkups

Ping workers every: seconds

Kick unresponsive workers after: 30 seconds

FIGURE K.14: The workers settings page



API Authentication

Database location	/srv/auta/auth.db
Bcrypt work factor	13 ^{2}log rounds
Token length	512 bits
Remove old tokens every	5 minutes
Allowed origins (CORS only)	https://autatest.f00f.nl/ https://autatest.f00f.nl http://localhost:8080 http://localhost:8081

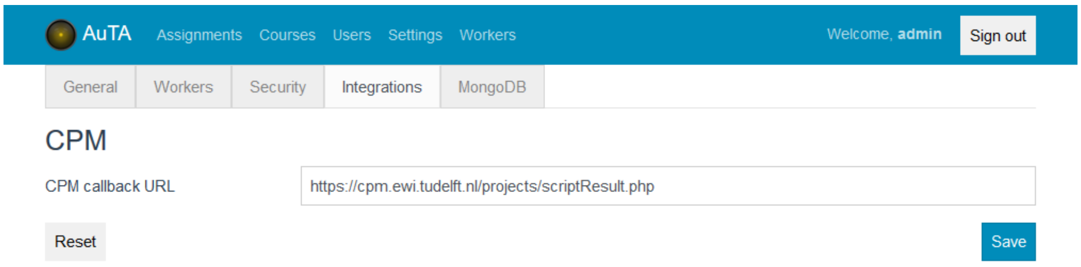
Assurance

Perform security checks every	30 minutes

Worker communication

Encryption key length	256 bits

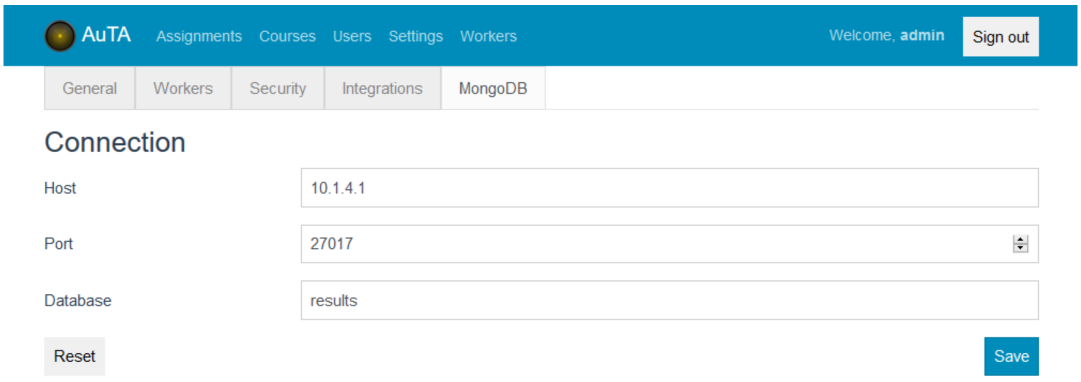
FIGURE K.15: The security settings page



CPM

CPM callback URL	https://cpm.ewi.tudelft.nl/projects/scriptResult.php

FIGURE K.16: The integrations settings page



Connection

Host	10.1.4.1
Port	27017
Database	results

FIGURE K.17: The MongoDB settings page

Appendix L

Repositories

In this appendix there is a list of repositories that were used for the creation of AuTA. The reason that most of the repositories are not present on the ewi GitLab server is that it would take a lot of time to request a new repository. We will hand over the repositories to the university at the end of the project. After that, they will be hosted on the same GitLab server as the main part of AuTA.

L.1 Links

- **Code-Measures** - https://gitlab.ewi.tudelft.nl/eip/code-measures
- **Trampoline** - https://git.wukl.net/f00f/trampoline
- **UI** - https://git.wukl.net/f00f/auta-ui
- **AuTA.js** - https://git.wukl.net/f00f/auta.js
- **Lizard** - https://git.wukl.net/f00f/lizard
- **J2V8** - https://git.wukl.net/f00f/auta-j2v8-plugin
- **Script runtime facade** - https://git.wukl.net/f00f/auta-script-runtime-facade
- **Test smells** - https://git.wukl.net/f00f/TestSmellDetector
- **Build environment** - https://git.wukl.net/f00f/auta-build-env

Appendix M

Research phase

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
> 50	untestable, very high risk

(A) Cyclomatic Complexity risk category assignment

	maximum relative LOC		
rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
--	-	-	-

(B) Cyclomatic Complexity system ranking calculation

ISO 9126 maintainability	source code properties: volume	complexity per unit	duplication	unit size	unit testing	
	++	--	-	-	o	
analysability	x		x	x	x	o
changeability		x	x			-
stability					x	o
testability		x		x	x	-

(C) Mapping source code properties to maintainability

FIGURE M.1: Calculating maintainability score figures, from [19]

M.1 Interviews

M.1.0.1 Dr. Santosh Ilamparuthi - EEMCS Data Steward

The EEMCS faculty data steward was approached with a few questions regarding the usage and management of student data. The programming assignments students deliver may contain personally identifiable information such as student number, email address or name. Care has to be taken to maintain the integrity of the data, and ensure the data is protected against unauthorized change or access.

The following topics were discussed:

- 3rd party tools - In order to analyze certain metrics, third-party tools can be used. Since potentially sensitive data is being handled (student code submissions), we need to make sure that we know how these tools use this data. Data should not be sent to other servers. This means that all third-party tools must be audited or isolated to make sure the data remains within the university's servers.
- Minimize identifiable conditions - Under no condition should students be able to see or identify other student assignments. This also applies to developers — they should not be able to access any student results, perhaps only the status of the server. Lecturers and TAs should be the only people to have access to student results for their courses.
- Data lifetime - It should be possible to remove students and their assignments from the results database. It is not clear how long data is allowed to be stored. This is being looked into.
- Data access requests - According to the GDPR, students have the right to access all personal data concerning them. The tool should be able to retrieve all data regarding a specific student.
- Communication - Communication between different services on the TU Delft LAN happens behind a firewall. Communication should be done at (at least) the same level of security as all other communication.
- Data Storage - The same as above applies — data storage should be done at (at least) the same level of security as other data storage.

M.1.0.2 Thomas Overklift - Head TA

To get a view of what a TA would require from the tool we interviewed Thomas Overklift. He is a Head TA for a number of courses in the Computer Science Bachelor Programme. Thomas indicated that a different approach should be used for smaller assignments such as OOP assignments and larger projects such as the Context Project or Bachelor Thesis Projects. Feedback should be more detailed for smaller projects and should give useful tips to improve the students' style and general programming skills. There should be a trade-off between the amount of time needed to configure the tool to give more detailed feedback and the quality of feedback given.

For larger projects, the tool would probably produce less meaningful feedback. For example, the tool might flag a method with 7 parameters for having too many parameters. However, if the method uses dependency injection this might be a better, more readable solution than splitting this method into smaller sections. The tool could be used to flag certain areas of code that may be problematic for TAs or instructors to review. The tool could also give feedback to students that these areas of code may need to be looked at.

Both Thomas and Frank Mulder also mentioned that an AI could be developed to create more complex feedback, this would, however, be something for the future and not something that would be possible in a single bachelor project.

M.1.0.3 Frank Mulder - Instructor

Frank Mulder is an instructor for Object-Oriented Programming (OOP) and Introduction to Python Programming (IPP). IPP is a course that is aimed at students that do not study Computer Science. The course is an introduction to programming, and therefore code metrics are less important. Although code quality is important, it is more important that students understand how programming works in general. Feedback such as: `"cyclomatic complexity for method foo() > 10"` is less relevant. Feedback needs to be more human-readable and could be adjusted to give programming advice to students to improve their quality and skills.

WebLab integration is very important for this instructor. At the moment, WebLab only gives information about which spec-tests pass and which fail. If a pass fails, a stack trace can be passed back to the student. This is not always easy to understand and is less relevant for the IPP course for students who have no experience with programming. WebLab could, for example, send the student's code to an AuTA API endpoint, which then returns more readable feedback for students.

M.1.0.4 Maurício Aniche - Instructor

Maurício Aniche is a lecturer for the Software Quality and Testing course that is given in the fourth quarter of the first year. During this course, students are expected to have some general knowledge about programming and are learning how to test their code accordingly. As the level of skill is different for this course compared, there are also different requirements for AuTA. Maurício came up with three different things that he would like to see in the tool:

First he wanted metrics that were specifically for test code as metrics for production code and test code are different. These metrics are described in the paper by Nachiappan Nagappan [2].

Secondly, he would like to give students feedback per metric how they performed relative to their cohorts because this would motivate them to improve on that area if they performed sub-par. An overview of these statistics should be visible by the lecturer to quickly view which groups are doing great work and which groups need to invest a bit more time to pass the course. This would be somewhat similar to the way SIG classifies projects. In the dashboard, it will also be possible to view the full reports that the students received. The lecturer will also be able to view which students will still have a running build and on which worker this is running. Only the admin will be able to change anything related to the configuration of AuTA.
Thirdly, it would be nice to have a way for students to locally run their code through the tool as with the current system students are being punished if they have a couple of failing builds in a row, something which they can not check if it will happen if the tool is only triggered after each commit. If we were to implement the first two points, AuTA could be considered for use in the course next year.

M.1.0.5 Daniel Pelsmaeker / Danny Groenewegen / Elmer van Chastelet - WebLab

WebLab is a learning management system used for a number of courses at the TU Delft. Students are able to complete programming assignments in WebLab. Unit tests are then used to calculate a score for student assignments.

We spoke with the developers of WebLab, as many stakeholders would like to see our tool and WebLab integrated. They requested that we send them our API documentation so that they could build something in WebLab that would allow a submission to be send to our tool. They would create a button that would allow students to request feedback from our tool. This would not run at the same time as WebLab's unit tests, as this would overload AuTA with messages.
When we start integrating AuTA into WebLab we start with only one AuTA assignment for every WebLab submission, later on we might be able to provide an assignment per language or even per course. They also requested that we would generate course specific API tokens so that other courses can't be accessed if the right token is not present.
Lastly they requested that we return the results of our tool in JSON format instead of our HTML generated report so that they can parse the output of our tool themselves.

M.1.0.6 Annibale Panichella - Project Coach

We had weekly meetings with Annibale Panichella, who may also use our tool next year. As a coach, he recommended several aspects that we should pay attention to during the project and some features that he would expect to see from such a tool.

First, he mentioned that we should test the security of our system; the system will be used by a lot of people who might not always have the best intentions in mind. We should assume that all users can be malicious, and should secure our system as such.

He also mentioned that there are certain simple messages that we could return as feedback, which are also better than directly returning the value of the metric. For example, when a class has too many parameters in the constructor we could reply with that it might be wise to refactor the class into multiple classes.

M.1.0.7 Otto Visser - Client

We also discussed our list requirements with our client. He added an extra requirement to the list as he would like to be able to grade assignments based on their submission. This would also mean that an instructor should be able to assign partial grades to certain metrics.

Appendix N

Stress Test Results

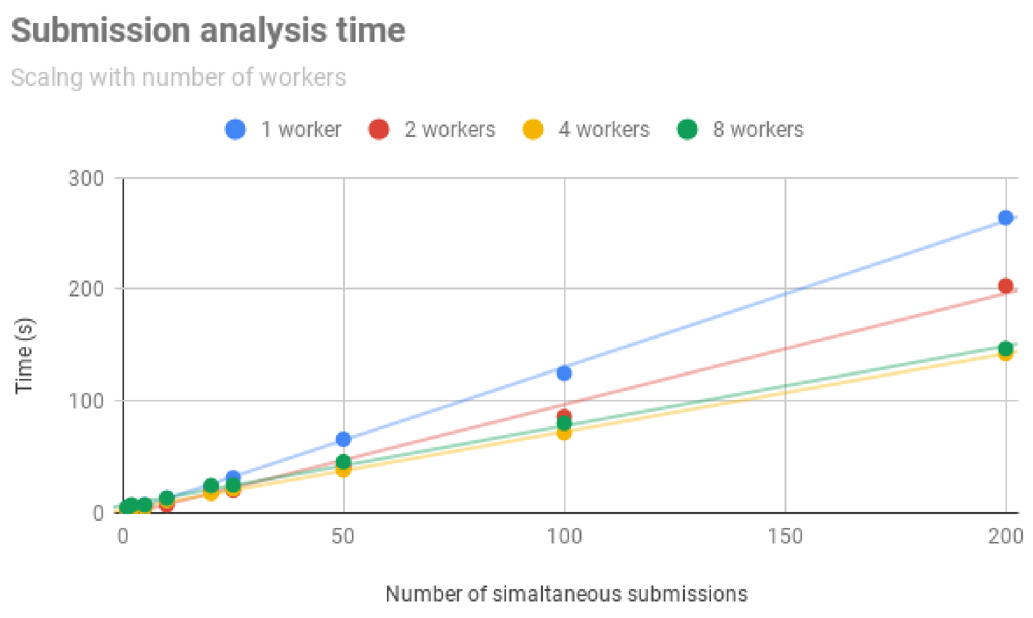


FIGURE N.1: Graph of total time with varying number of workers and submissions

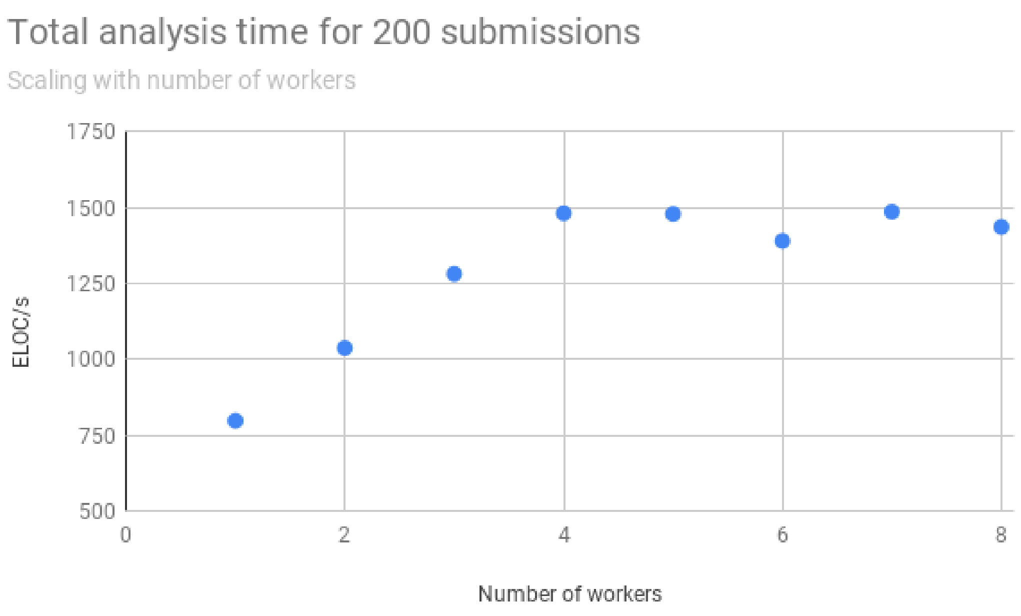


FIGURE N.2: Graph of effective lines of code per second to analyze 200 simultaneous submissions with varying number of workers

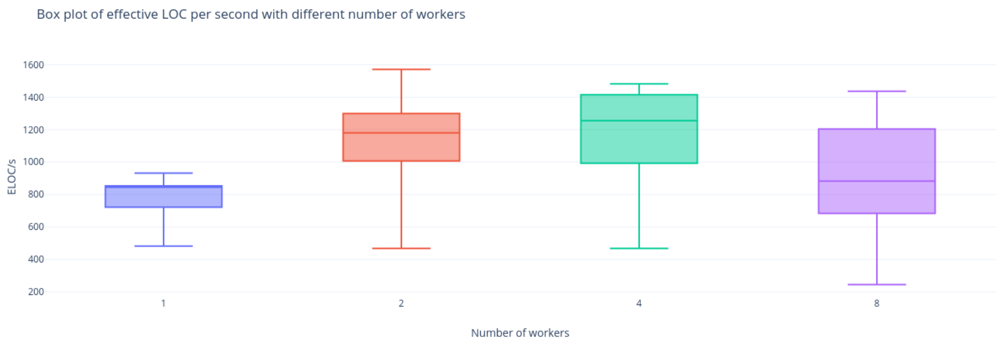


FIGURE N.3: Boxplot of effective LOC/s with varying number of workers

Appendix O

SIG feedback

O.1 First feedback

De code van het systeem scoort 4.0 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Interfacing en Unit Size.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Dit kan worden opgelost door parameter-objecten te introduceren, waarbij een aantal logischerwijs bij elkaar horende parameters in een nieuw object wordt ondergebracht. Dit geldt ook voor constructors met een groot aantal parameters, dit kan een reden zijn om de datastructuur op te splitsen in een aantal datastructuren. Als een constructor bijvoorbeeld acht parameters heeft die logischerwijs in twee groepen van vier parameters bestaan, is het logisch om twee nieuwe objecten te introduceren.

Voorbeelden in jullie project:
- CPMController.uploadAction(MultipartFile, String, String, String, String, String, String, String, String)
- PyLintResult.PyLintResult(String, String, String, String, String, String, String, String, String)
- MessageReceiver.MessageReceiver(WorkerPool, Jump, ResultsProcessor, Threads, GlobalSettings, SubmissionRepository)

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:
- Core.main(String[])
- AuthDbInitializer.init(DatabaseConnection)

- CKAnalyzer.getClassLevelMetrics(CKClassResult, Entity)
- AttAsmAnalyzer.getPassingScripts()

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

Appendix P

Infosheet

The infosheet is included as the next page

Automated Teaching Assistant (AuTA) 2.0

TU Delft Educational Innovation Projects **Presentation date:** 04/07/2019

Description

The number of enrollments in the Computer Science programme at the TU Delft has increased dramatically over the last few years. In order to reduce the workload of TAs and instructors, the team members developed a tool that can be used to automatically generate feedback for student assignments based on static and dynamic analysis results.

The main challenge during the project was designing a highly scalable and reliable tool that can generate useful feedback for students. During our research phase, we found a number of different education tools that generate feedback for students based on code analysis. They highlighted the importance of formative feedback, where students are provided with advice on how to improve their code.

The process began by interviewing multiple stakeholders at the TU Delft to gather a list of requirements. These stakeholders included multiple instructors and a head TA. An agile SCRUM methodology was used throughout the project to be able to adapt to a change in requirements from any of the stakeholders.

Product

AuTA uses various other tools (command line, java libraries, python tools) to calculate metrics for files or entire projects. Using the UI, instructors or TAs can specify what criteria and metrics are used to generate warnings, failures or tips for a student's submission. These results are formatted in a report and sent to students, who can use the feedback in the report to improve their code. TAs can view more detailed information in our UI to identify possible problematic areas in the code.

AuTA was end-to-end tested using Gitlab CI, unit tested, and manually tested throughout the project.

The product is deployable as is, but could use further development to improve the ease with which instructors and TAs can set up assignments for a course. The product will first be used for courses in September 2019.

Luc Everse	**Erik Oudsen**	**Tim van der Horst**	**Ewoud Ruighaver**
luc@wukl.net	e.oudsen@gmail.com	timvanderhorst@gmail.com	ewoud.ruig@gmail.com
Interests			
Software Quality and Engineering, UX, OS development	Machine Learning, Big data	Software pattern design, data analysis, networking	Cloud computing, Machine Learning
Contributions			
UI, exports, user-defined suppressions, SAML Single-Sign on, Qualified name generation	Qualified name generation, analyzer tools integration, aggregated analyzer, report	UI, metric histograms, data model overhaul, Qualified name generation, external service integration	Scaling with kubernetes, language auto-detection, dynamic analysis

Client: Ir. Otto Visser - Distributed Systems Group - TU Delft
Coach: Dr. Annibale Panichella - Software Engineering Research Group

The final report for this project can be found at: http://repository.tudelft.nl

Bibliography

[1] C. van Uffelen, “Aantal eerstejaars technische informatica ‘valt reuze mee’”, *TU Delta - TU Delft*, Sep. 2018. [Online]. Available: `https://www.delta.tudelft.nl/article/aantal-eerstejaars-technische-informatica-valt-reuze-mee` (visited on Apr. 23, 2019).

[2] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, “Early estimation of software quality using in-process testing metrics: A controlled case study”, *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, May 2005, ISSN: 0163-5948. DOI: `10.1145/1082983.1083304`. [Online]. Available: `http://doi.acm.org/10.1145/1082983.1083304`.

[3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design”, *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] F. B. Abreu and R. Carapuça, “Object-oriented software engineering: Measuring and controlling the development process”, in *Proceedings of the 4th international conference on software quality*, vol. 186, 1994, pp. 1–8.

[5] D. Ufer and R. Hilton, *JaSoMe: Java source metrics build status*, Jan. 2017. [Online]. Available: `https://github.com/rodhilton/jasome` (visited on May 2, 2019).

[6] T. Yin, *Lizard*, Jan. 2012. [Online]. Available: `https://github.com/terryyin/lizard/` (visited on May 2, 2019).

[7] M. Lacchia and J. Sargiotto, *Radon*, Sep. 2012. [Online]. Available: `https://github.com/rubik/radon` (visited on May 2, 2019).

[8] InfoEther, *PMD*, 2003. [Online]. Available: `https://pmd.github.io/` (visited on May 2, 2019).

[9] SpotBugs authors, *Spotbugs*, Oct. 2017. [Online]. Available: `https://spotbugs.github.io/` (visited on May 2, 2019).

[10] Checkstyle authors, *Checkstyle*, 2001. [Online]. Available: `http://checkstyle.sourceforge.net/` (visited on May 2, 2019).

[11] H. Keuning, J. Jeuring, and B. Heeren, “A systematic literature review of automated feedback generation for programming exercises”, *ACM Trans. Comput. Educ.*, vol. 19, no. 1, 3:1–3:43, Sep. 2018, ISSN: 1946-6226. DOI: `10.1145/3231711`. [Online]. Available: `http://doi.acm.org/10.1145/3231711`.

[12] V. J. Shute, “Focus on formative feedback”, *Review of Educational Research*, vol. 78, no. 1, pp. 153–189, 2008. DOI: `10.3102/0034654307313795`. eprint: `https://doi.org/10.3102/0034654307313795`. [Online]. Available: `https://doi.org/10.3102/0034654307313795`.

[13] M. Goedicke, M. Striewe, and M. Balz, “Computer aided assessments and programming exercises with jack”, eng, Essen, ICB-Research Report 28, 2008. [Online]. Available: `http://hdl.handle.net/10419/58160`.

[14] P. Antonucci, C. Estler, D. Nikolić, M. Piccioni, and B. Meyer, "An incremental hint system for automated programming assignments", in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '15, Vilnius, Lithuania: ACM, 2015, pp. 320–325, ISBN: 978-1-4503-3440-2. DOI: `10.1145/2729094.2742607`. [Online]. Available: `http://doi.acm.org/10.1145/2729094.2742607`.

[15] T. A. Majchrzak and C. A. Usener, "Evaluating the synergies of integrating e-assessment and software testing", in *Information Systems Development*, R. Pooley, J. Coady, C. Schneider, H. Linger, C. Barry, and M. Lang, Eds., New York, NY: Springer New York, 2013, pp. 179–193, ISBN: 978-1-4614-4951-5.

[16] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart, "Learning feedback in intelligent tutoring systems", *KI - Künstliche Intelligenz*, vol. 29, no. 4, pp. 413–418, Nov. 2015, ISSN: 1610-1987. DOI: `10.1007/s13218-015-0367-y`. [Online]. Available: `https://doi.org/10.1007/s13218-015-0367-y`.

[17] T. Murray, S. Blessing, and S. Ainsworth, *Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive and intelligent educational software*. Springer Science & Business Media, 2003.

[18] N. Truong, P. Bancroft, and P. Roe, "Learning to program through the web", in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '05, Caparica, Portugal: ACM, 2005, pp. 9–13, ISBN: 1-59593-024-8. DOI: `10.1145/1067445.1067452`. [Online]. Available: `http://doi.acm.org/10.1145/1067445.1067452`.

[19] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability", Oct. 2007, pp. 30 –39, ISBN: 978-0-7695-2948-6. DOI: `10.1109/QUATIC.2007.8`.

[20] J. P. Correia and J. Visser, "Benchmarking technical quality of software products", in *2008 15th Working Conference on Reverse Engineering*, Oct. 2008, pp. 297–300. DOI: `10.1109/WCRE.2008.16`.

[21] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data", in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10. DOI: `10.1109/ICSM.2010.5609747`.

[22] J. P. Correia and J. Visser, "Certification of technical quality of software products", in *Proc. of the Int'l Workshop on Foundations and Techniques for Open Source Software Certification*, 2008.

[23] Software Improvement Group, *Better Code Hub*, 2016. [Online]. Available: `https://bettercodehub.com/` (visited on Apr. 26, 2019).

[24] SonarSource, *Sonarqube product website*, 2018. [Online]. Available: `https://www.sonarqube.org/` (visited on Jun. 26, 2019).

[25] CodeClimate, *Velocity pricing*, 2019. [Online]. Available: `https://codeclimate.com/velocity/pricing/` (visited on May 2, 2019).

[26] Pivotal, *RabbitMQ*, 2007. [Online]. Available: `https://www.rabbitmq.com/` (visited on May 2, 2019).

[27] iMatix, *ZeroMQ*, 2013. [Online]. Available: `https://www.zeromq.org/` (visited on May 2, 2019).

[28] Cloud Native Computing Foundation, *Kubernetes*, Jun. 2014. [Online]. Available: `https://kubernetes.io/` (visited on May 2, 2019).

[29] The Kubernetes Authors, *Deployments - scaling a deployment*, May 2018. [Online]. Available: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment` (visited on May 2, 2019).

[30] Department of Software Engineering of Rochester Institute of Technology, *Software unit test smells*, Feb. 2018. [Online]. Available: `https://testsmells.github.io/` (visited on May 1, 2019).

[31] M. Aniche, *CK*, Oct. 2015. [Online]. Available: `https://github.com/mauricioaniche/ck` (visited on May 1, 2019).

[32] SURF, *Labrador*, 2019. [Online]. Available: `https://www.surf.nl/stimuleringsregeling-open-en-online-onderwijs/projecten-stimuleringsregeling-pijler-online` (visited on May 3, 2019).

[33] A. Roques, *PlantUML*, 2019. [Online]. Available: `https://github.com/plantuml/plantuml` (visited on May 15, 2019).

[34] EclipseSource, *J2v8*, 2014. [Online]. Available: `https://github.com/eclipsesource/J2V8` (visited on Jun. 26, 2019).

[35] Google Inc., *V8 javascript engine*, 2008. [Online]. Available: `https://github.com/v8/v8` (visited on Jun. 26, 2019).

[36] Mozilla, *Rhino*, 2012. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino` (visited on Jun. 26, 2019).

[37] Oracle, *Project nashorn*, 2013. [Online]. Available: `https://openjdk.java.net/projects/nashorn/` (visited on Jun. 26, 2019).

[38] OASIS, *Assertions and Protocols for the OASIS Security Assertion Markup Language(SAML) V2.0*, Mar. 2005. [Online]. Available: `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf` (visited on Jun. 24, 2019).

[39] ——, *Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0*, Mar. 2005. [Online]. Available: `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf` (visited on Jun. 24, 2019).

[40] ——, *Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0*, Mar. 2005. [Online]. Available: `http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf` (visited on Jun. 24, 2019).

[41] M. Wahl, "A summary of the X.500(96) User Schema for use with LDAPv3", RFC Editor, RFC 2256, Dec. 1997, `http://www.rfc-editor.org/rfc/rfc2256.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc2256.txt`.

[42] M. Smith, "Definition of the inetOrgPerson LDAP object class", RFC Editor, RFC 2798, Apr. 2000.

[43] Internet2, *Registrations in the `urn:mace:dir:attribute-def` namespace*, Jul. 2003. [Online]. Available: `https://www.internet2.edu/products-services/trust-identity/mace-registries/urnmace-namespace/urn-mace-dir-registry/urn-mace-dir-attribute-def/` (visited on Jun. 24, 2019).

[44] "Advanced encryption standard (AES)", Tech. Rep., Nov. 2001. DOI: `10.6028/nist.fips.197`. [Online]. Available: `https://doi.org/10.6028/nist.fips.197`.

[45] N. I. of Standards and T. (NIST), *Announcing approval of federal information processing standard (fips) 197, advanced encryption standard (aes)*, Dec. 2001. (visited on Jun. 25, 2019).

[46] S. Vaarala, A. Nuopponen, and T. Virtanen, "Attacking predictable IPsec ESPInitialization vectors", in *Information and Communications Security*, vol. 4th International Conference, ICICS 2002, Dec. 2002, pp. 160 –172.

[47] T. L. Alves and J. Visser, "Static estimation of test coverage", in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sep. 2009, pp. 55–64. DOI: `10.1109/SCAM.2009.15`.

[48] Python Software Foundation, *Full grammar specification*, Jun. 2019. [Online]. Available: `https://docs.python.org/3.6/reference/grammar.html` (visited on Jun. 25, 2019).

[49] github, *Linguist*, 2011. [Online]. Available: `https://github.com/github/linguist` (visited on Jun. 25, 2019).

[50] S. Bailliez, N. K. Barozzi, J. Bergeron, S. Bodewig, P. Chanezon, J. D. Davidson, T. Dimock, P. Donald, D. Gillard, E. Hatcher, D. Holt, B. Kelly, M. Kruithof, A. J. Kuiper, A. Lévy-Lambert, C. MacNeill, J. Matèrne, S. Mazzocchi, E. Meade, S. Ruby, N. Seessle, J. S. Stevens, W. Siberski, M. Umasankar, R. Vaughn, D. Walend, P. Wells, C. Wilhelms, and C. Strong, *Directory-based tasks*, The Apache Software Foundation, May 2019. [Online]. Available: `https://ant.apache.org/manual/dirtasks.html` (visited on Jun. 25, 2019).

[51] Faculty and students of the Department of Software Engineering of Rochester Institute of Technology New York, *Test smells*, 2019. [Online]. Available: `https://testsmells.github.io/pages/about.html` (visited on May 15, 2019).

[52] P. Hammant, *QDox*, 2019. [Online]. Available: `https://github.com/paul-hammant/qdox` (visited on May 15, 2019).

[53] PyCQA, *Pylint*, 2019. [Online]. Available: `https://github.com/PyCQA/pylint` (visited on May 23, 2019).

[54] V. Zhou, *Profanity checker*, 2019. [Online]. Available: `https://github.com/vzhou842/profanity-check/` (visited on Jun. 20, 2019).

[55] ——, *Profanity checker*, 2019. [Online]. Available: `https://victorzhou.com/blog/better-profanity-detection-with-scikit-learn/` (visited on Jun. 20, 2019).

[56] B. M. Hill, *Clbuttic*, Jun. 2008. [Online]. Available: `https://revealingerrors.com/clbuttic` (visited on Jun. 25, 2019).

[57] Shibboleth Consortium, *OpenSAML 3*, 2011. [Online]. Available: `https://wiki.shibboleth.net/confluence/display/OS30/Home` (visited on Jun. 24, 2019).

[58] SpringSource, *Spring SAML 2.0 plugin*, 2019. [Online]. Available: `https://mvnrepository.com/artifact/org.springframework.security.extensions/spring-security-saml2-core/1.0.9.RELEASE` (visited on Jun. 24, 2019).

[59] Shibboleth Consortium, *OpenSAML 2*, 2008. [Online]. Available: `https://wiki.shibboleth.net/confluence/display/OpenSAML` (visited on Jun. 24, 2019).

[60] ——, *Service Provider*, 2017. [Online]. Available: `https://www.shibboleth.net/products/service-provider/` (visited on Jun. 24, 2019).

[61] A. Klaver, T. Kinkhorst, R. Teeuwen, T. Fransen, A. Terpstra, B. Zoetekouw, N. van Dijk, H. Bekker, and F. Morsch, *Attributes in surfconext*, Feb. 2013. [Online]. Available: `https://wiki.surfnet.nl/display/surfconextdev/Attributes+in+SURFconext` (visited on Jun. 26, 2019).

[62] No formal list of connected providers could be found, but the SURFconext "select an institution" page lists most, if not all, institutions.

[63] A. Walker, A. Pipinellis, A. Caiazza, bikebilly, C. Ho, D. Gruesso, D. Planella, D. Zaporozhets, D. Maan, D. Griffith, E. Alcántara, E. Eastwood, E. Read, F. Busatto, G. L. Breton, G. Roulot, G. Bizon, J. Fargher, J. D., J. Lambert, K. Trzciński, M. Amirault, M. Ramos, M. Fletcher, M. Čupić, M. Cabrera, M. Lewis, N. Pestelos, O. Gonzalez, P. Slaughter, R. Dickenson, S. McGivern, Sergej, S. Hu, S. Moore, T. Chupryna, T. Kuah, Tiger, T. Zallmann, T. Maczukin, T. Jaquith, walkafwalka, and Z.-J. van de Weg, *Auto DevOps*, GitLab Inc. [Online]. Available: `https://docs.gitlab.com/ee/topics/autodevops/` (visited on Jun. 27, 2019).

[64] *@Nullable and @NotNull*, JetBrains s.r.o. [Online]. Available: `https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html` (visited on Jun. 27, 2019).

[65] *@Contract*, JetBrains s.r.o. [Online]. Available: `https://www.jetbrains.com/help/idea/contract-annotations.html` (visited on Jun. 27, 2019).

[66] E. Bouwers, J. Visser, and A. van Deursen, "Getting what you measure", *Commun. ACM*, vol. 55, no. 7, pp. 54–59, Jul. 2012, ISSN: 0001-0782. DOI: `10.1145/2209249.2209266`. [Online]. Available: `http://doi.acm.org.tudelft.idm.oclc.org/10.1145/2209249.2209266`.

[67] T. Parr, *Definitive ANTLR 4 reference*, S. D. Pfalzer, Ed. The pragmatic bookshelf, 2012.