

Architecture Design

Code Measurement for Mass Education

Delft University of Technology

EEMCS

Luc Everse	4470397
Tim van der Horst	4437837
Erik Oudsen	4573706
Ewoud Ruighaver	4577159
Bailey Tjong	4474686

Contents

1 Introduction	1
1.1 Design goals	1
2 Software architecture views	2
2.1 Subsystem decomposition	2
2.2 Hardware/software mapping	3
2.3 Persistent data management	3
2.3.1 Database	3
2.4 Concurrency	4
2.4.1 API	4
2.4.2 Core-Worker communication	4
2.5 Metrics	5
2.5.1 Tools	5
3 Glossary	6
A Subsystem/component diagram	7
B Metrics and supported languages	8
C System flow	9

1 Introduction

This document provides an architecture overview of the system built as part of the Education context project. The first section focuses on what the design aims to accomplish, while the second section describes the architecture itself.

1.1 Design goals

This section lists the primary design goals for this project.

Scalability

Considering the high number of students next year and the fact that students tend to submit as close to the deadline as humanly possible, the system will need to be able to cope with extremely high loads where at least one thousand submissions arrive within the same second. The verification process should not suffer from such peaks so the system can maintain a fair, first-come-first-serve contract.

Availability

Automated checking should become part of the workflow for signing off homework, so it is very important that the service remains available. It goes without saying that

we will take great care in developing the service, but even the best efforts cannot prevent all contingencies. Therefore the service will be split up into multiple microservices that may fail and be restarted without (greatly) affecting the availability of the whole system.

Flexibility

Because no two courses are the same, the system should provide different metrics and measurements for different usecases. Furthermore, we also want to provide a general continuous-integration service so courses can run acceptance tests.

Security

Since a TA's judgement may be decided based on the service's output, it is of great importance that the service is secure so that no student can affect their own verdict other than by passing the assignment's criteria. That entails two goals: a secure front-end and a secure environment for submission to execute in.

2 Software architecture views

This section describes the design choices made to fulfill the design goals from the previous section.

2.1 Subsystem decomposition

As shown in Appendix A, the product is subdivided into four main packages. The flow of the system is shown in Appendix C.

Core service

The main part of the service, this provides the API for the entire application and distributes jobs among the workers. This part is intended to be lightweight in order to be able to accept a huge number of jobs within a short time.

Worker

One or more instances of this service run the actual checks and tests on the submissions. Having multiple separate workers, perhaps even on separate hardware, allows for concurrent and therefore more responsive results.

Report store

This is a document store where all results and reports end up in. Using an existing document-oriented database allows us to cheaply implement complex aggregations.

Management interface

An interface preferably developed in cooperation with the fraud group, in order to make administration easier.

The management interface communicates with the core server over standard HTTP over a well-documented API so other services can interface with it as well. HTTP was chosen because it is an established protocol with universal support.

Workers interface with the core service over a custom bidirectional protocol with minimal overhead, as the only data sent over this channel is a config object describing how to analyze which assignment and analysis results should be returned.

2.2 Hardware/software mapping

Each of the above subsystems are intended to be run as separate processes, especially the worker instances, which are intended to run on different servers (virtually or physically).

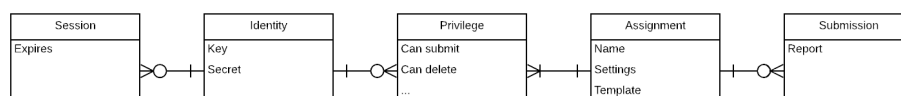
The services themselves are written in Java 1.10, chosen for its solid platform and extensive collection of libraries; learning a different language or writing these libraries by ourselves would simply cost too many man-hours. For the same reason the core service will use the Spring Boot web application framework: setting up a server and modifying it later on during the lifetime of the product is trivial.

Another benefit of Java is that the code is mostly platform-independent, removing the need to worry about the system the service will eventually run on.

Dynamic analysis will run in a containerized environment to prevent submissions from interfering with the service, which must remain secure. Existing containerization platforms are also extremely customizable, so assignments can define special environments for the submissions to run in.

Docker will be used as the primary containerization provider. The main alternative was LinuxContainers, however the project does not provide a centralized image registry, adding yet another component to be maintained. More traditional chroot jails do not provide enough customizability out of the box, let alone any premade images. Meanwhile, full virtual machines are bulky and slow to start, however they may be used in the future if complete isolation becomes necessary.

2.3 Persistent data management



Service configuration information such as API credentials and what checks to run for which assignment are stored in a relational database.

Code metric reports are put in a separate document-oriented database to make generating aggregate reports easier.

2.3.1 Database

Using a document-store instead of a relational database reduces the overhead of mapping Results objects to store them in a database. The Results object can have

arbitrary attributes (e.g. lines of code, cyclomatic complexity). In the future, more attributes may be added. Additionally, some attributes are language specific. Storing an object whose structure is almost certain to change in the future is much easier when using a non-relational database compared to a relational database, where the schema must be defined beforehand.

Spring has built-in support for MongoDB, which makes it very simple to attach to our tool simply by creating a container class for our results class (which stores the ID and results) and annotating the classes.

We will index the Submissions collection on AssignmentId, and Results object on SubmissionId. This will speed up queries such as "all Submissions with id=AssignmentId". Another option would be to store SubmissionIds in the Assignment objects. However, this would mean the Assignment object has to be updated whenever a Submission object is inserted.

2.4 Concurrency

2.4.1 API

As described above, the main subsystem of the product is the Core service. The Core service presents an API which can be found at <https://auta.f00f.nl/api/>. This API is the main and only end-user interface with the service, which will allow other services to be integrated with the product in a uniform way.

2.4.2 Core-Worker communication

The Core service needs to be able to communicate with each instance of a Worker. The actual messaging happens through a custom protocol for message passing: the Jump protocol. Jump is slightly simpler than most message queues, having only eight bytes of overhead (4 bytes of length, repeated at the end as a sanity check). No additional integrity or security checks are performed, as the protocol assumes that the underlying layers are safe and optionally secure.

2.5 Metrics

2.5.1 Tools

To be able to calculate metrics for various different languages, we have chosen to make use of a mixture of existing third-party tools and libraries and custom implementations where we couldn't find any or it was trivially easy to create them ourselves. Our complete list of current metrics with their supported languages can be found in Appendix B.

The way we chose these tools depended on the metrics they could provide and how up to date the tools were. We started off by creating some simple metrics for proof of concept. These metrics were line length and lines of code. We choose for these metrics as they were easy to create and easy to check by hand. This way we were able to check if our tool was giving the correct results.

After we had decided that the structure of our product was working we started implementing bigger third party tools for the more complex metrics. These included for example cyclomatic complexity and assertions per method. We choose for these metrics as they would be the most useful for the described use cases at the start of the project. The table in Appendix B shows an overview of which tools/libraries are used to calculate which metrics.

We have chosen to use Lizard¹ to calculate cyclomatic complexity and eLOC metrics. Lizard is a tool which can be used to calculate cyclomatic complexity and eLOC for C++14, C, Java, Scala and Python. It is a tool written in Python. We include the Lizard source code as a .zip and run the code using Jython² so we do not have to call the library using its command line interface. Jython is a lightweight Java implementation of Python which can be used to run Python code in the JVM.

QDox³ was used to parse and analyze Java in order to analyze Javadoc and test classes.

ANTLR was used to get metric results from Assembly submissions. The grammar used in this ANTLR file has been specifically created to support the needs of the project. Using this grammar we can check for comment ratios and check whether recursion was implemented or not.

We decided not to use CKJM, a tool for Java which calculates Chidamber and Kemerer [1] and other metrics for Java. This tool was recommended to us after the midterm meeting. It requires the compiled .class files of a project. It is used to calculate class level metrics, which are not always applicable to the code this tool will be used to analyze. In addition, it is not kept up to date (last commit was 3 years ago)

¹<https://github.com/terryyin/lizard>

²<http://www.jython.org/>

³<https://github.com/paul-hammant/qdox>

3 Glossary

API *Application Programming Interface*

A "contract" between software components describing how to communicate.

Containerization

An application of *virtualization* where a server provides multiple isolated environments in which (sets of) programs can run without interfering.

Container image

A template for containers.

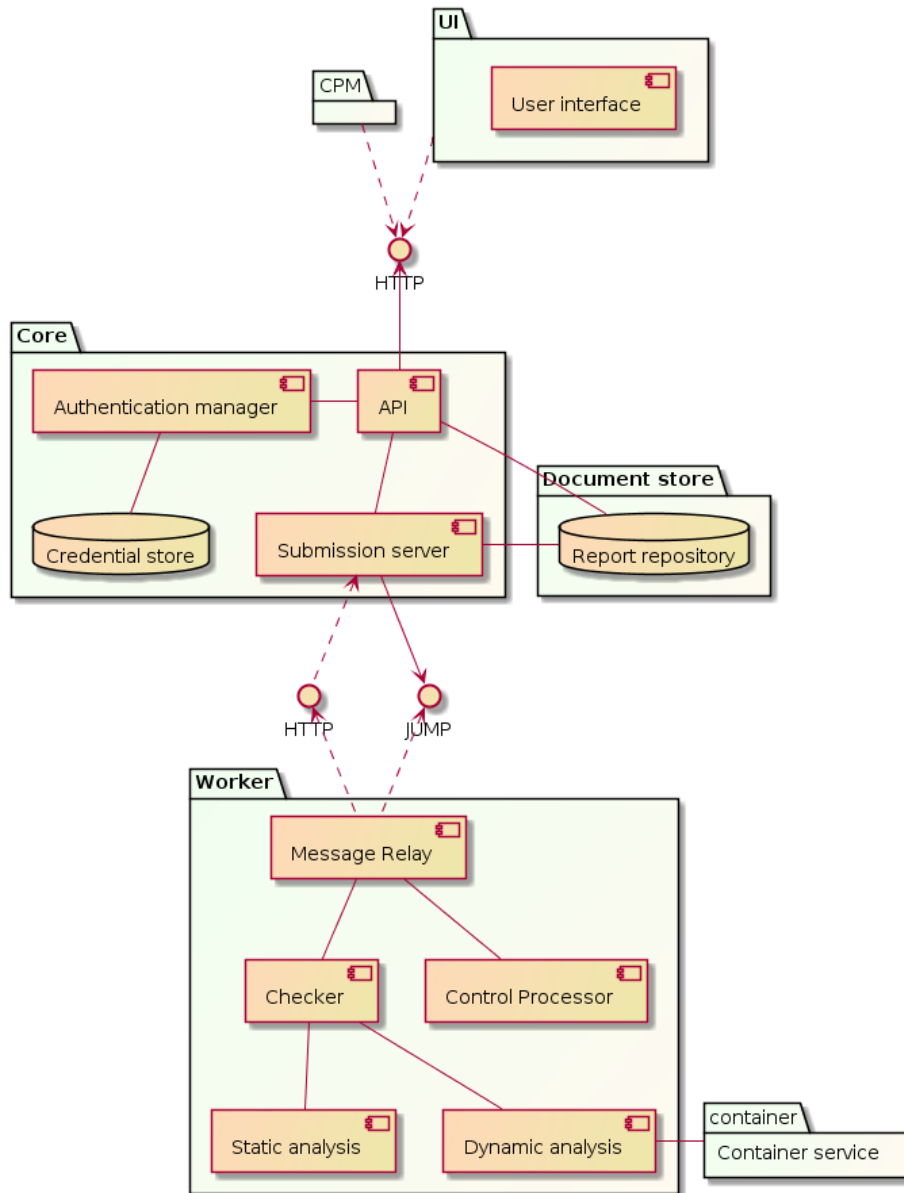
eLOC

Effective lines of code, which is the number of lines that are not comments or blank.

Image registry

An application where container images are stored. Some services provide a publicly-accessible registry.

A Subsystem/component diagram

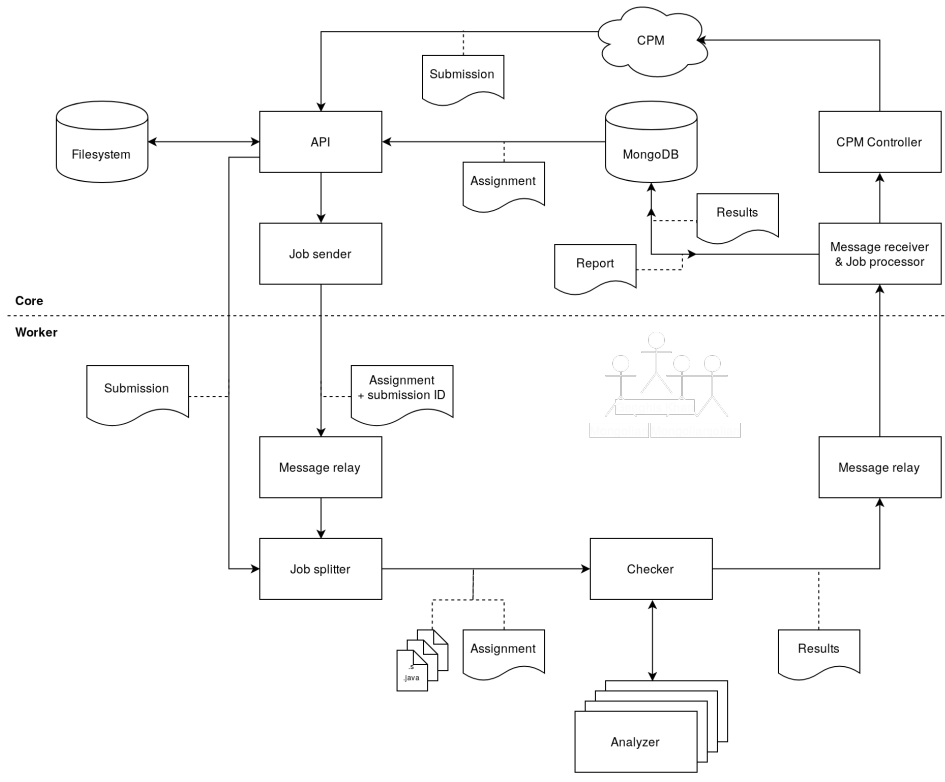


B Metrics and supported languages

Table 1: Metrics with their supported languages

metric	language
Cyclomatic complexity	Java Python Scala C C++
Method Length	Java Python Scala C C++
Line Length	Generic
Comment ratio	Java C C++ Python Assembly
UML	Java
Docker	Generic
Javadoc existence	Java
Javadoc violations	Java
Parameter Count	Java
Assertions per test	Java
Method count	Java
Test Method count	Java
Field count	Java
Constructor count	Java
Recursion	Assembly
Lines of code	Java C C++

C System flow



References

- [1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.