1 Coding and Integration

After a brief overview of the project and the project plan, this section will focus on the main technologies used in the project and the rationale behind choosing them. Moving on, we will discuss how these components were integrated and eventually deployed.

1.1 Overview

One of the first decision we have made this term was to completely recreate our prototype of Thalia. Firstly, this daring move was the consequence of a new implementation decision, see in more detail 1.4.1. Secondly, as prototypes are meant to be disposable and are designed only to answer key questions about the system [?], the proof of concept served its purpose, giving us a chance to refine the structure and the quality of the code.

Despite of the risks posed by using an Software Version Control (SVC) Host such as GitHub, we have decided to continue using as our software development platform. The reasoning behind this is as mentioned in our Technical Report, that our Data Processing Module is a completely separate component in our system which none of the other components is able to access. Additionally, we are storing API keys as environment variables in a file that is not tracked by our SVC system, which minimizes the probability of us exposing sensitive data.

TODO talk about API keys and security measures

We have also decided to develop the application with python as our main choice of programming language. Even though this choice seemed obvious from the beginning, we did consider its main benefits, these would be the following:

- Python is a high level programming language allowing us to better focus on the application.
- A standard choice for prototyping.
- Provides superb third party libraries and frameworks for free.
- Easy to integrate if we were to choose other languages at some point in our development.
- The whole team was already familiar with the language, saving us the precious time needed to learn another programming language.
- Our application does not require an unreasonable amount of computation, so there is no need for a more efficient programming language such as C [?].

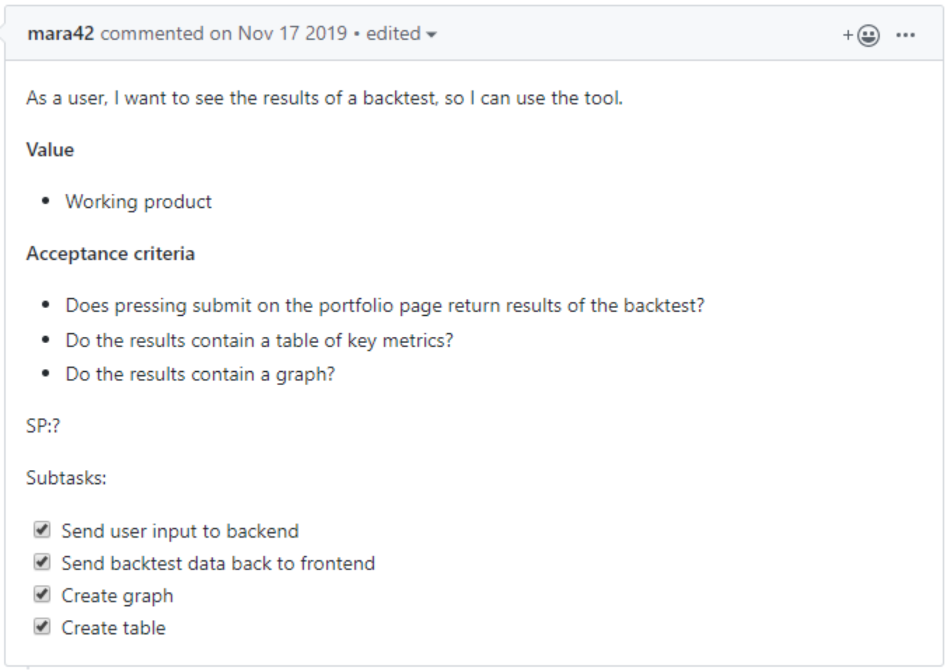
We will discuss other technologies used in more detail after the discussion on project planning.

1.2 Planning

Early in the inception phase of development we have decided that our goal was not to have fixed responsibilities in our team, allowing everybody to work on each component of the system. This decision has also eased the code review process, as we had no status differences to distort the error-correcting mechanism [?], and allowed us to direct comments at the code and not the author [?]. For these reasons we have decided to follow the Egalitarian Team structure, and make use of the flexibility offered by it.

Our workflow was centered around the tools provided by GitHub. We used a ticketing system to divide and distribute tasks among team members. These tickets were sometimes given by the team on the weekly meetings, but occasionally they were chosen by the member proposing the feature or change. Our goal with this approach was to divide larger jobs into smaller tasks.

A typical ticket in our project was an encapsulation of a user story, as it consisted of a title, a one liner, value, acceptance criteria, and sub tasks. In the first half of development we also used effort-oriented metrics, called story points to measure the amount of work but we decided to abandon this aspect. An example of a ticket can be seen on Figure1.


mara42 commented on Nov 17 2019 • edited

As a user, I want to see the results of a backtest, so I can use the tool.

Value

- Working product

Acceptance criteria

- Does pressing submit on the portfolio page return results of the backtest?
- Do the results contain a table of key metrics?
- Do the results contain a graph?

SP:?

Subtasks:

- [x] Send user input to backend
- [x] Send backtest data back to frontend
- [x] Create graph
- [x] Create table

Figure 1: Ticket Example

TODO update git stats

Throughout the whole project we had a total of 110 tickets, and 71 pull requests. The following graph also shows the number of contributions to master, excluding merge commits.

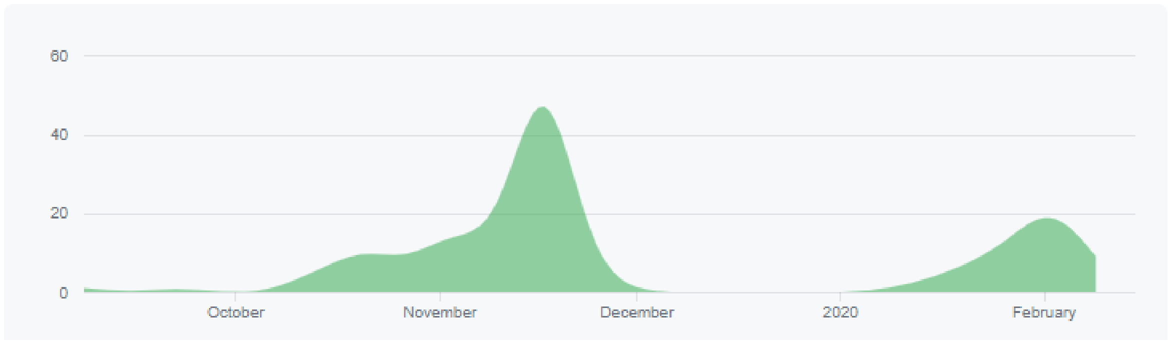


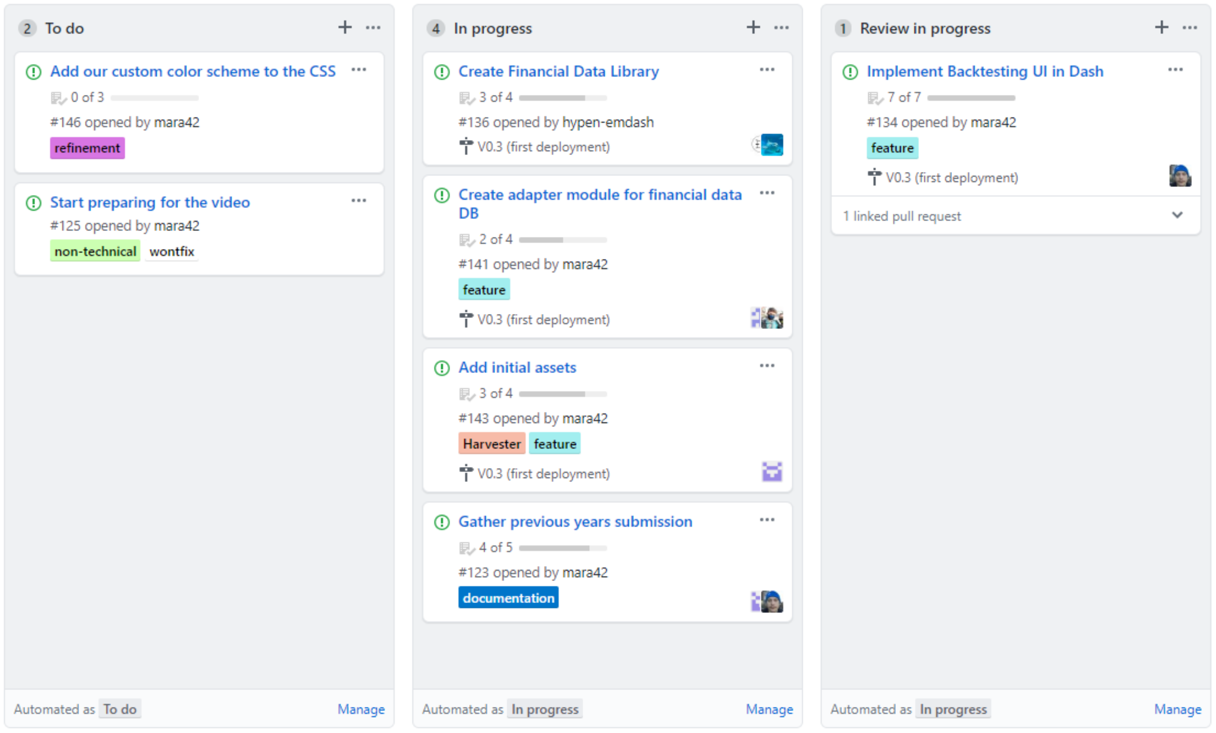
Figure 2: Contributions to the Master branch

Nevertheless, this was not always achievable especially in the beginning of the development when more crucial components of the systems were developed. In these cases, we assigned the ticket to a pair, or group of people. This approach achieved the following:

- Improved the overall code quality and fastened production [**?**].
- Minimised review time on the long run.
- Distributed the knowledge of large system components amongst a few people instead of one.
- Eased introducing the new team member to the project.

In addition, we also had a scrum board as an overview for the ongoing tickets. Although in the Scrum community there are ongoing discussions about the benefits of a physical Scrum board over an online one

[?], given no actual workplace this was not possible to achieve. This allowed us to see which tickets needed to be reviewed and which were ready to be merged. The tickets/ cards were distributed into columns, such as To do, In progress, Review in progress, Review complete and Finished. A truncated picture of this scrum board can be seen on figure Figure3.


2 To do

Add our custom color scheme to the CSS
0 of 3
#146 opened by mara42
refinement

Start preparing for the video
#125 opened by mara42
non-technical wontfix

Automated as To do — Manage

4 In progress

Create Financial Data Library
3 of 4
#136 opened by hypen-emdash
V0.3 (first deployment)

Create adapter module for financial data DB
2 of 4
#141 opened by mara42
feature
V0.3 (first deployment)

Add initial assets
3 of 4
#143 opened by mara42
Harvester feature
V0.3 (first deployment)

Gather previous years submission
4 of 5
#123 opened by mara42
documentation

Automated as In progress — Manage

1 Review in progress

Implement Backtesting UI in Dash
7 of 7
#134 opened by mara42
feature
V0.3 (first deployment)
1 linked pull request

Automated as In progress — Manage

Figure 3: Scrum board - truncated

The workflow then largely depended on the task at hand and the person working on it. It was up to them if they met for pair-programming, or chose to work individually. In the beginning we all agreed on the developing environment, and coding standards, for a detailed analysis see the later section 1.5. Since this report was a relevant portion of the workload, we decided to treat it as code. The report was written in LaTeX, first creating the overall structure of the document with a main.tex compiling the sections together. This let us to work on different sections separately just like features in our software.

Regardless of the nature of the ticket, the week, or in case of some larger tasks, two weeks, ended by the creator indicating that the changes were ready to be integrated. This was done by publishing the changes, and creating a new pull request, pushing the changes onto the master branch. To ensure code quality, we set up a continuous integration (CI) environment, more on that in 1.5. After all checks have passed, the changes were successfully integrated into the code base.

1.3 Schedule

As mentioned in our Technical Report, we introduced some milestones, or releases for our product based on incremental sets of features. This was done to help us stay on track with development. The roadmap based on this schedule can be seen on Figure4.

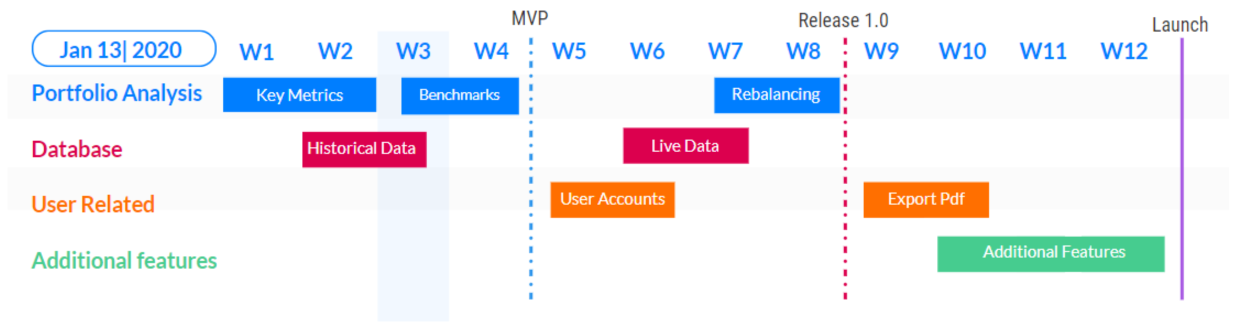


Figure 4: Technical Report: Roadmap - revisited

TODO retrospective on schedule

1.4 Key Implementation Decisions

Let us now discuss the main technologies used in our project, these can be categorised as follows:

1.4.1 Web Framework

Choosing the right web framework was more controversial than originally planed. The two major options for Python are Django and Flask. Although we decided to use Django for our MVP in the first semester, we had to spend a relevant portion of the time available learning the framework, so we had to decide whether we were going to stick with it or learn Flask. In the end, we opted to go with Flask for the following reasons:

- Django has one architecture that all projects must share, and we have designed the architecture for our project ourselves. While neither architecture is wrong, the two are not compatible. Flask, on the other hand, is structure-agnostic, so we can lay out the code as we see fit.
- Flask comes with the bare minimum for web-development, which means that we don't need to manage the complexity of any feature we're not using. Django has a more complete feature-set from the beginning. This would be desirable in a large web application, but introduces significant overhead in our case, where the website has only a handful of pages.
- Django all but insists on using its ORM for all database interaction, while we plan to have a more manual approach.
- Our concerns were also confirmed by more experienced web-developers, suggesting simpler alternatives.

As stated this decision has contributed to the opportunity to recreate the basis of our application, and to apply what we have learnt from our prototype.

1.4.2 GUI

1.4.3 Business Logic

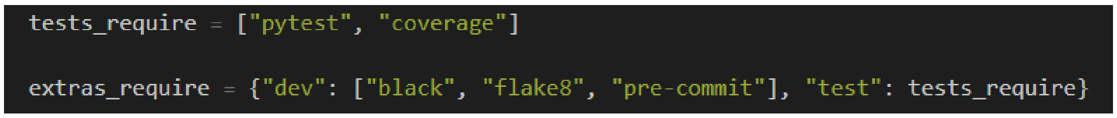
1.4.4 Database & Finda

1.4.5 Harvester

1.5 Integration

Writing well documented and good quality code is one thing, but making sure it all works together as a whole is a completely different story. In the first term, many hours have been wasted on trying to integrate different components of the system which did not want to fit together. Even then we had some DevOps tools in place [?], but considering that we had to produce significantly less code and more documentation last term, this was not in our focus.

Starting afresh the coming term, we have decided to set up the development environment again. Upon opening the setup.py file, we see a list of required libraries, and the following two lines of code:



```
tests_require = ["pytest", "coverage"]

extras_require = {"dev": ["black", "flake8", "pre-commit"], "test": tests_require}
```

Figure 5: setup.py - Development environment

One of the first decisions we had to make, is to decide on a standards coding style. This is exactly what flake8 is for, which we can see amongst the extras on Figure5. The original documentation of flake8 defines it as " [...]is a command-line utility for enforcing style consistency across Python projects. By default it includes lint checks provided by the PyFlakes project, PEP-0008 inspired style checks provided by the PyCodeStyle project, and McCabe complexity checking provided by the McCabe project" [?]. However, as many other developers we also decided to redefine the maximum line-length from 79 to 88 as we found this convention a hindrance.

The next tool also enforcing a uniform style, was the black auto-formatter for python [?], which formatted the code for us upon every save if enabled, and also when commitin code. This has been achieved by the use of githooks [?], which are programs that are triggered upon certain git actions, for this we needed the pre-commit package for setting up these actions, and write the configuration file. This ensured that both flake8 and black have been run before publishing changes.

1.5.1 Continuous Integration

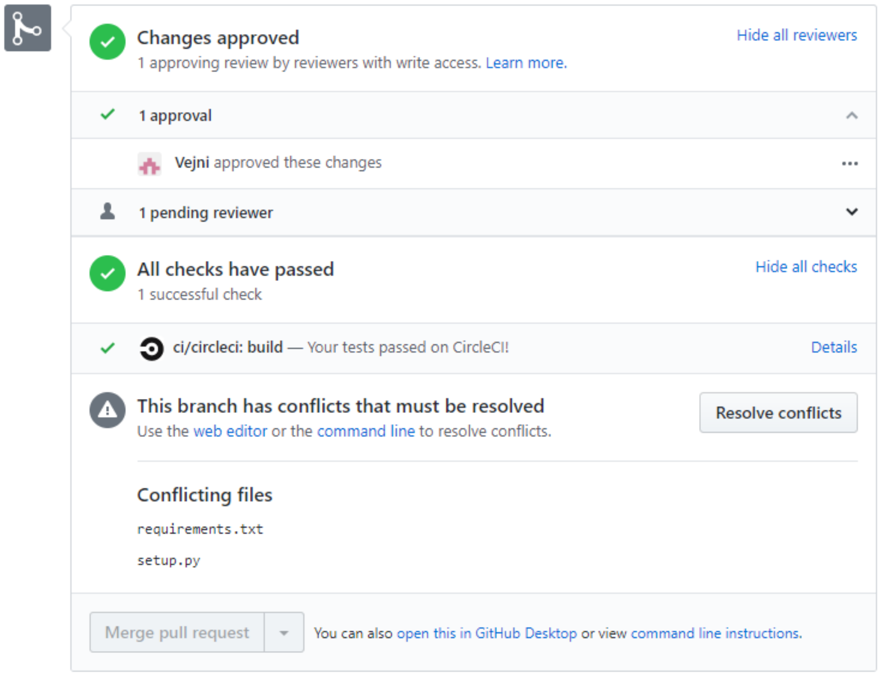
Many studies have investigated the positive effects of developing in a continuous integration (CI) environment [?], [?]. Regardless of what these are exactly, its obvious benefit is that it provides security and uniformity for projects. We already made some steps to achieve a uniform style, but had no means to know whether the code published has been also passed its tests. Note in this section we will only discuss testing as a part of CI and not the testing strategy, for that see the section ??.

Another important part of the DevOps toolchain is the use of containers, which is what Docker helps to achieve [?]. Docker helps developers focus on writing code rather than worrying about the system the application will be running on, and also helps to see dependency and library issues. As Docker is open source, there are many free to use docker images available online [?]. When choosing the CI environment Docker support was one of the main requirements.

The most promising candidate for this was CircleCI, which is a cloud-based system with first-class docker support and a free trial. After connecting our GitHub repository to CircleCI, and setting up a configuration, CircleCI now does the following on every pull-request:

1. Sets up a clean container or virtual machine for the code.
2. Checks previously cached requirements.
3. Installs the requirements from requirements.txt
4. Caches the requirements for faster performance.
5. Clones the branch needed to be merged.
6. Runs flake8 one last time and saves results.
7. Runs tests and saves results.
8. Deploys the master branch to Heroku, see 1.6

The outcome of these steps is visible on CircleCI, but more importantly also on GitHub, and it refuses to merge if failing test (or no tests) have been found.


Changes approved
1 approving review by reviewers with write access. Learn more.

1 approval

Vejni approved these changes

1 pending reviewer

All checks have passed
1 successful check

ci/circleci: build — Your tests passed on CircleCI!

This branch has conflicts that must be resolved
Use the web editor or the command line to resolve conflicts.

Conflicting files

```
requirements.txt
setup.py
```

You can also open this in GitHub Desktop or view command line instructions.

Figure 6: CircleCI on GitHub

With the help of these steps and CircleCI we managed to ensure that the code written is uniform and of good quality. It significantly reduced the time needed for integrating and code reviewing. The last step was now to deploy the system.

1.6 Deployment