Thalia Project Report

Team Charlie

Martti Aukia, Albert Boehm, Arthur-Louis Heath, George Stoian, Daniel Joffe, Weronika Kakavou, Marcell Veiner

University of Aberdeen
Sunday 8th March, 2020

Acknowledgement

TO DO: Thank stackoverflow and everyone else here.

Contents

1 Introduction

1.1 Project Overview

The aim of this project was to create a portfolio backtesting software, which enables creating custom portfolios and measuring their performance with different backtesting functions. The user can pick from a variety of assets, some of which are Equities, Fixed Income, Currencies, Commodities, and Cryptocurrencies. Risk metrics and performance are then visualized for the given asset allocation.

1.2 Motivation/Rationale

Since retail investing is a growing market, our target audience consists of individual investors, who instead of seeking the full package that comes with financial advising, would like to take over the wheel and assess the viability of their investment strategies themselves. As retail investors are non-professionals and invest comparatively small amounts, with financial advice services being non affordable for those individual clients, our goal was to create a product that would not only be more affordable for small retail investors, but would also include a variety of international assets, which most existing backtesting software fails to provide [**?**].

1.3 Project management strategy

The creators of Thalia are:

- Martti Aukia, *Team Leader*
- Arthur-Louis Heath, *Deputy Team Leader*
- Albert Boehm, *Chief Editor*
- Marcell Veiner
- George Stoian
- Daniel Joffe
- Weronika Kakavou

Our goal was the creation of a high quality industrial prototype of the Thalia backtesting software based on the identified project requirements [**?**] and optional features [**?**]. The team held regular meetings, both with the project guide, Dr Nigel Beacham, and the course coordinator Dr Ernesto Compatangelo. During the analysis stage meetings took place weekly and were aimed to discuss ideas and requirements. Whereas later, during the implementation stage the team held meeting at least once per week, some of which were aimed to discuss the design for the tool with the inclusion of some coding sessions.

As in the past[**?**], our workflow was centered around the GitHub platform and the tools it provides. Furthermore, we continued to follow the Egalitarian Team structure, as this worked very well during the first term of the course. In our Technical Report [**?**] we also discussed the use of effort-oriented metrics, i.e. story points, which were assigned to tickets based on the time needed and the functionality of the task. However, the following term we concluded that in many occasions these metrics failed to successfully measure the size of a task, as they were artificially assigned. Based on this observation, we decided not to make use of them for the rest of the development.

As we were fortunate enough to gain an additional member this term, some of our initial effort was focused on introducing her to the project and team dynamics. Given our access to a large team, it was common to see coding tickets assigned to pairs and groups rather than individuals. We believe this not only resulted in writing better-quality code, but in faster team communication when making design decisions.

1.4 Budget

As previously discussed, we did not have any budget restrictions other than time. All of us agreed to allocating a minimum of 10 hours per week to development efforts and discussed personal expectations well in advance. We estimate the value of our products goodwill to be approximately £13,000. We base this estimate on the developer time allocated to this project, the total duration of 20 weeks, and average developer salaries in the UK[**?**] (taking into account that we have no industry experience, meaning the value of our work is realistically towards the lower end of the range).

2 Background and Competitors

As a result of our initial competitor analysis, we may group the competing software into two categories. For a comprehensive list of available backtesters please see [**?**]. The first consists of various third-party trading software, such as Fidelity [**?**], MetaTrader [**?**] and NinjaTrader [**?**] that offer backtesting features as well. Services in this category include features such as buying and selling of financial assets, prediction of future prices and storing and managing of users' money. Although it may be beneficial to consider some features related to real time trading, these would only be part of a future development process, and are not in the scope of our prototype.

The second category consists of pieces of software that do not offer trading as a service, and solely focus on backtesting. Thus, for now, we shall only consider the feature set provided by the second category. In the following paragraphs we will consider some of the design decisions made by competing software, together with the brief analysis and conclusion on each feature.

2.1 Portfolio Visualizer

Our main competitor of the second category is an online backtesting tool named Portfolio Visualizer[**?**]. During the inception phase of development we heavily relied on this website for writing the requirement analysis. Let us now briefly dissect what it has to offer.

Upon opening the website we are greeted with a brief description of the domain, together with the following input form:

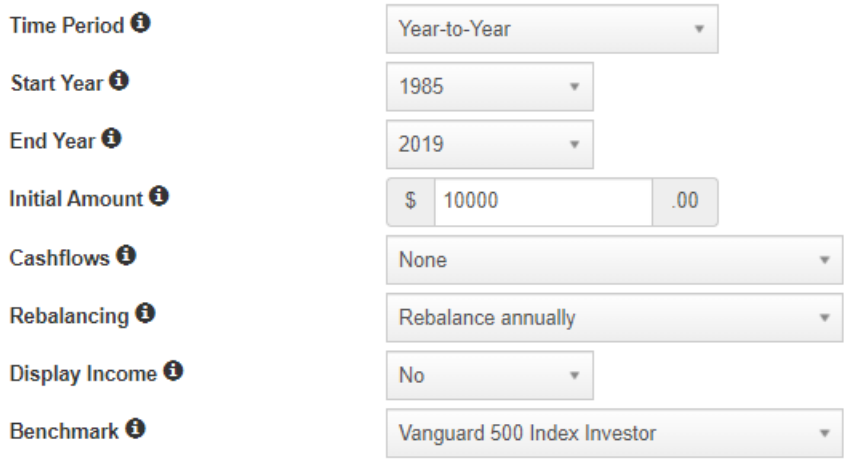


Figure 1: Portfolio Visualizer - Input (source: https://www.portfoliovisualizer.com/)

As we can already see, the key elements of portfolio analysis are provided. Considering the functionality of our prototype as a baseline (that is testing an allocation of assets with a fixed initial investment on a fixed time period), we see that in addition users are able to do the following set of features:

- Set the endpoints of the investment period, up to months.
- Set the initial investment.
- Specify a regular cashflow and its frequency.
- Select a rebalancing strategy.
- Select a benchmark strategy for comparison.
- Compare multiple strategies at the same time.
- Adjust to inflation.

- Select from a set of lazy portfolios.
- Calculate additional metrics.
- Export the results to PDF, Excel, or save link.

At first it may seem as if Portfolio Visualizer meets all the requirements needed for a financial backtester, and indeed our main criticism is regarding the UI, responsiveness and user experience, and is a result of the initial user testing.

The UI design is simplistic and has a non-commercial, bare-bones look, and although this was appreciated while testing the system it certainly does not improve the user experience. The input, mostly using dropdown menus is straightforward to use, except for the selection of Assets, which we will discuss briefly at the end of this section.

Moreover, users will want to fine-tune their investment strategies, by changing their allocations frequently. The only means to do this using Portfolio Visualizer is to scroll to the top, change the input and rerun the entire simulation. Our goal is to design a more responsive and dynamic system, to ease this procedure.

The website also has user accounts, however little to no data is associated with a user's profile, which further decreases the overall user experience. Finally, as a last remark, which holds for most of the backtesting tools we have tried, the website is heavily US biased, and so the selection of asset classes is limited. Furthermore, we have no means of changing the currency, which would also be an important feature for a product accessible through the web (and therefore available to users from all over the globe).

2.2 Other Competing Software

The rest of the alternatives are of a significantly lower quality. In the remaining parts of this section we will briefly consider a few design decisions made by these websites.

Tree-like Asset Selection

One challenging aspect of designing such a system is the following:

> What is the most intuitive way for selecting a portfolio item?

Where a portfolio item could be: an equity, ETF index, a commodity, bond or stock. Within these classes we have more options to choose from. Many backtesters opted for a search bar, which is a sensible approach, but poor in practice. Many portfolio items are named similarly, and for a new user it is a barrier, as they might not know what is available.

One of the better option is what ETFReplay [**?**] has implemented, which is a tree-like selection form, shown below.

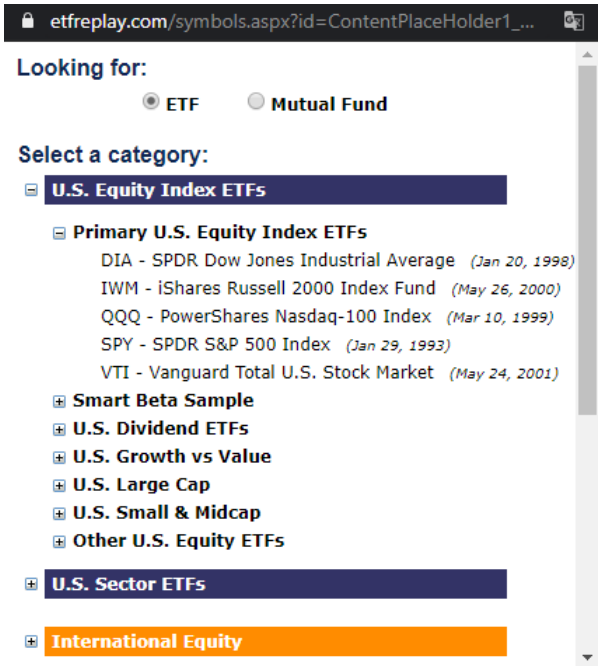


Figure 2: ETFReplay - Tree-like Structure (source: https://www.etfreplay.com/)

We feel this was the most intuitive to use, and will thus pursue a similar approach. The only potential issue associated with this is that it opens in a new window, which we would prefer to avoid.

Tiles

It may be worth briefly discussing an example of a backtester with a good layout design. We found the simplistic and tiled design of Backtest Curvo [**?**] a good choice as it gives the website an overall modern and fresh look, whereas most backtesters looked old and out of date. Our goal is to achieve a layout similar to this. For a further analysis on design decisions, please see section 4 of this report.

Simple Logic

In our whitepaper, we briefly discussed implementing a simple scripting language allowing users to simulate dynamic trading strategies [**?**]. As mentioned, this would not be in the scope of the course, but for inspiration we can have a look at the approach of Stockbacktest [**?**].

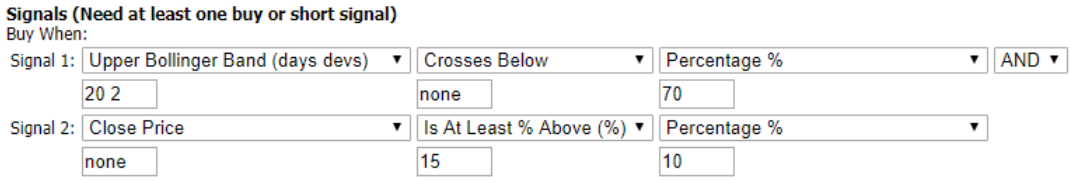


Figure 3: Stockbacktest - Simple Logic (source: http://stockbacktest.com/)

Although not intuitive at all, given its presentation, it shows this feature is also possible. We believe that after nailing down how exactly the user would create the rules, this would be a straightforward task, as the calculations involved are not more complicated than in the static case. As mentioned above, this is something we would like to implement in the future.

2.3 Summary

As we initially tested the competing software during the inception phase of development, we were left with three key observations. These were the following:

- Transparency: Many of the testers' webpages did not give the impression of being operated by a reputable businesses. In the worst case, they gave the impression of being untransparent or untrustworthy (misspelt words, advertisements with questionable subject matter), which is something we should avoid.
- Learning Curve: After getting to know our target audience we concluded that most seem willing to learn how to use a complex system if it is worth it.
- Design of the UI: It was easy to tell which backtesters are still getting updated, by looking at their design. We should aim at looking fresh. In addition, some backtesters offered so many features that every corner of their UI was packed with information, this is something we should also avoid doing.

The analysis was conducted informally with the help of colleagues studying or involved in finance. This allowed us to develop a more neutral and accurate impression, despite the subjective nature of the observations. Having analysed the competing software, we are now better suited to determine the requirements of our system.

3 Requirements

Earlier in the inception phase of the development, we classified our requirements using the established FURPS+ model [**?**]. Since identifying requirements for our initial project report [**?**] we have also identified some additional requirements that Thalia should fulfil based on feedback from our project guide. Let us briefly revisit our main functional and non-functional requirements. The items outlined below focus on some of the key requirements we have so far identified as features necessary for providing a compelling product for paying customers.

3.1 Functional Requirements

Portfolio Configuration	
Allocate fixed amount/proportions of the portfolio to given assets	Choose how much each asset contributes to the portfolio's total value using either percentages or raw monetary amounts
Find assets quickly by category or name	When adding an asset the user can search a category for assets or search for a specific asset by its name
Share portfolio	Portfolios can be shared between people using a URL
Edit portfolio	Change asset allocation and their distributions in a portfolio

Portfolio analysis	
Compare portfolios	Use multiple portfolios in a single analysis to see differences in their performance
Use a selection of lazy portfolios	Select an existing common portfolio to compare against, such as common index funds (e.g. Vanguard 500 Index Investor or SPY)
Plot portfolio as a time-series	View portfolio performance as a line graph for a quick overview
Specify a time frame for the analysis	Select start and end dates for portfolio analysis
Choose rebalancing strategy	Optionally choose a strategy for buying and selling assets to meet your strategy e.g. buying and selling stocks each year to ensure the value of portfolio stays at 60% stocks and 40% bonds (i.e. maintain the initial allocation)
Change the distribution of assets in a portfolio using a slider	A slider for each asset to quickly increase or decrease its proportion of the total value
Edit portfolio analysis	Change parameters for portfolio's analysis after running it (e.g. date range or rebalancing strategy)

View results	
See key numerical figures	Show important numerical metrics for a portfolio's performance such as Initial Balance, Standard Deviation, Worst Year, Sharpe Ratio, and Sortino Ratio
See both real and nominal values	See portfolio's value as both adjusted and not adjusted for inflation
A breakdown of portfolio value at specific points of time	See what the value of the portfolio is at some point in time (e.g. January 3rd 1997)
Export result of analysis	Exports results to PDF for sharing and offline reading

User accounts	
Combine portfolios	Combine two portfolios' assets into one single portfolio
Save portfolio analysis for later	Save portfolio analysis parameters to the account so you can rerun it with a single click
Delete saved portfolio analysis	Remove a stored portfolio analysis from your account
Manage portfolio analyses	Edit saved portfolio analysis with different assets, distributions or other parameters
Sign-up, log in and log out	Basic authentication

Assets	
Choose assets from European market	Data for European assets were found to be lacking in competing products
Choose assets from Equities, Fixed Income, Currencies, Commodities, and Cryptocurrencies	Coverage of some of the largest asset classes

3.2 Non-functional Requirements

1. Usability:
   - The product must be easily usable for users who already have some financial investment experience.
   - The basic backtesting interface needs to look familiar to people already experienced with it.
   - The product must have detailed instructions on how to use its advertised functions.
   - All major functions must be visible from the initial landing page.
   - Must work in both desktop and mobile browsers.
   - The results page should scale with mobile.
2. Reliability:
   - The product must have a greater than 99% uptime.
   - All our assets need to have up to date daily data where the asset is still publicly tradeable.
   - All assets supported by the system must provide all publicly available historical data.
3. Performance:
   - The website should load within 3 seconds on mobile [2].
   - Large portfolios must be supported - up to 300 different assets.
4. Implementation:

- The system needs to work on a cloud hosting provider.

5. Interfacing:
   - The Data Gathering Module must never use APIs stated to-be-deprecated within a month.
   - The Data Gathering Module must not exceed its contractual usage limits.
6. Operations:
   - An administrator on-call will be necessary for unexpected issues.
7. Packaging:
   - The product needs to work inside a Linux container (e.g. Docker).
   - All dependencies need to be installable with a single command.
8. Legal:
   - All user testing must be done with ethical approval from the University.
   - UI must display a clear legal disclaimer about the service not providing financial advice.
   - All third-party code should allow for commercial use without requiring source disclosure (e.g. no GPL-3).
   - User data handling should comply with GDPR.
   - Provided services should not constitute financial advice under UK law to avoid being subject to financial advice legislation and potential liability.
9. Accessibility
   - Display items should be clearly labelled.
   - UI should scale to accommodate different screen sizes and aspect ratios,
   - UI elements and text superimposed over one another should have high contrast in their colors.
   - UI should allow for the use of assistive technologies to accommodate individuals with accessibility issues.

In the literature, the FURPS+ model has been criticised for disregarding developer consideration [**?**], such as not taking into account portability and maintainability. Furthermore it has been pointed out, that our requirements fail to capture ...

TODO

3.3 Use Cases

TODO

4 Design

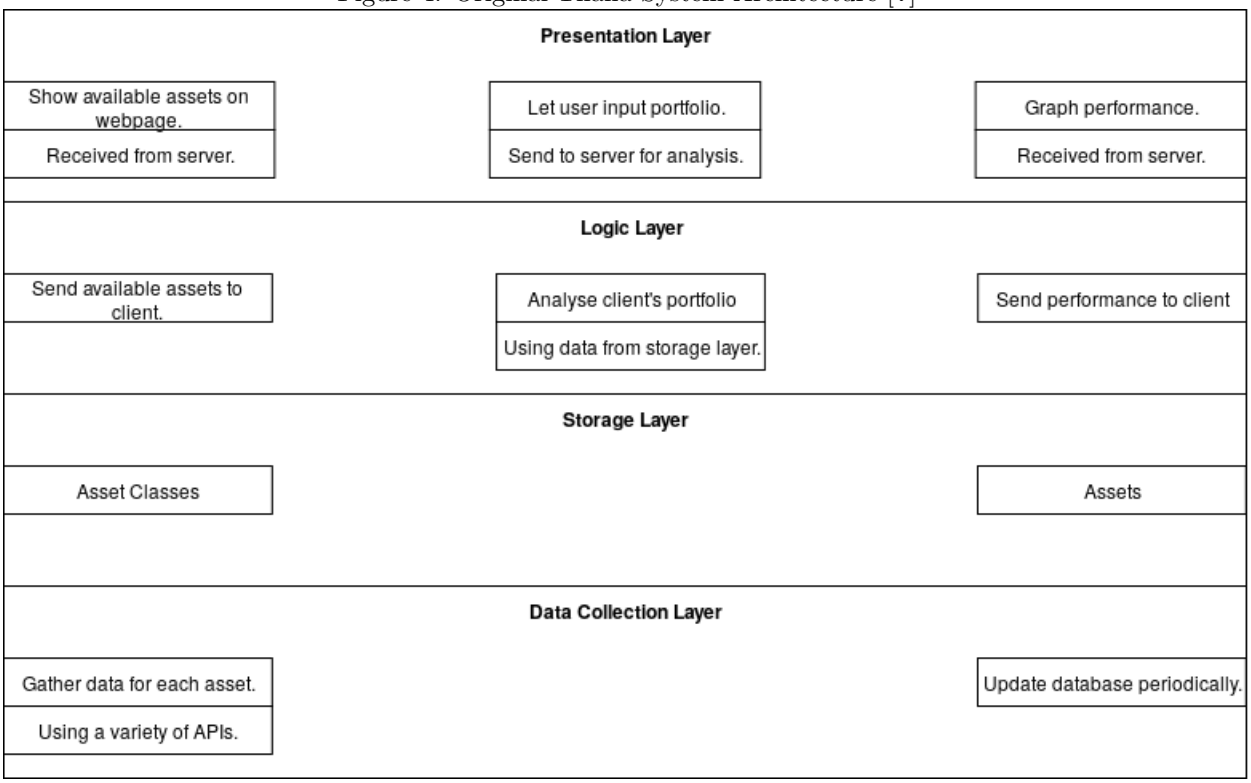
4.1 General Design Decisions

We have chosen pandas dataframes [**?**] as the common data format for both data exchange and any calculations. Pandas provides us with data structures "that cohere with the rest of the scientific Python stack" [**?**], such as NumPy, which we are using to calculate historical returns and risk metrics (4.4). Additionally, it is supported natively by many third party APIs and can even be used to read data from SQL databases. For additional information, please consult the official pandas documentation [**?**].

4.2 Overview of System Architecture

Before discussing any specific component and architectural layer in depth, it is worth revisiting our original architecture [**?**].

Figure 4: Original Thalia System Architecture [**?**]



The above schematic illustrates how we modified a typical Three Layer architecture [citation needed] to include a Data Collection Layer. Subsequent discussion will refer to this layer as the 'Data Harvester'. It consists of adapters for third party APIs which offer price data for financial assets. The associated API will be queried according to a configurable interval to allow for live updates to our price data. Additionally, initial seeding of the database with historical price data can also be achieved via the Data Harvester. The decision to isolate this system component has been made in order to increase security by limiting the acess of the Thalia web service to the financial data database to read-only and to allow for independent scaling of the Data Harvester and our web service [**?**].

While the architecture of our system has stayed the same, there have been some modifications to individual components. Most notable are the changes to the Database Structure examined in 4.6. The following sections will provide for a discussion of design decisions made on a layer-by-layer basis.

4.3 GUI Structure

4.4 Business Logic Structure

The responsibility of our business logic can be summarised as follows: Given an investment strategy specification input from a user via the GUI, retrieve relevant financial data from the database to perform calculations for the historical performance and associated risk metrics.

To achieve this, we have developed a library (Anda) that performs the necessary calculations. Anda is decoupled from both the presentation and database layer by relying on external providers for any price data and the specification of a strategy. This decision has been made to allow for alternative sources of price data in the future. One of our optional features for future development is allowing users to input their own price data for assets not supported by Thalia. This data could be uploaded, for example, as a CSV file or JSON. Without our current design, i.e. by coupling calculation of performance and metrics to database access, we would have to modify the business logic to support multiple data sources. Given our current implementation, however, we can simply parse the user data into a pandas dataframe in a wrapper around Anda and then call functions within the library as require. Currently supported metrics include Total Return, Max Drawdown, Best / Worst Year, and the Sharpe and Sortino Ratios. However, the library is open for extension, hence additional metrics may be added at any point.

For a closer look at how an investment strategy is specified, consider the following class that serves as input to Anda library functionality (e.g. for calculating the Sharpe Ratio [citation needed]):

```
import pandas as pd

class Strategy:
    def __init__(
        self,
        start_date: date,
        end_date: date,
        starting_balance: Decimal,
        assets: [Asset],
        contribution_dates,  # implements __contains__ for date
        contribution_amount: Decimal,
        rebalancing_dates,  # implements __contains__ for date
    ):
        self.dates = pd.date_range(start_date, end_date, freq="D")
        self.starting_balance = starting_balance
        self.assets = assets
        self.contribution_dates = contribution_dates
        self.contribution_amount = contribution_amount
        self.rebalancing_dates = rebalancing_dates
```

Listing 1: setup.py - Development environment

Here, Asset is a simple dataclass consisting of a ticker string (e.g. ‘MSFT‘ for Microsoft), a weight as a share of the portfolio overall (e.g. 0.25), a pandas dataframe holding historical price data ordered by date, and a pandas dataframe for dividends data (if any).

As alluded to earlier, functions within the library depend on a Strategy object for their calculations. For example:

```
def total_return(strat) -> pd.Series:
```

Listing 2: setup.py - Development environment

will calculate a series of total return values ordered by date within the date range specified in the passed Strategy instance.

Another important design decision has been the choice of data type to represent money, for example for price data. For this, we have chosen the Decimal type from the Python Standard Library decimal module, since it “provides support for fast correctly-rounded decimal floating point arithmetic“ [**?**]. As rounding errors and imprecision are unaccaptable for our application, using the Decimal type will allow us to reliably compute figures for prices, risk metrics, etc.

Finally, we have chosen NumPy [**?**] for performing numerical calculations as this allows for highly optimized computation through the use of vectorized operations.

4.5 Data Harvester Structure

4.6 Database Structure

4.7 Data Segregation

The decision was made early on to horizontally partition the data store by Thalia into two parts. One consisting of data related to users and user accounts and the other of financial data related to asset classes, assets and their historical prices. The following is the list of reasons the team documented for this decision:

- One alternative revenue stream we identified early on was the sale of our financial data as a separate product. This process would be trivially easy if it was stored in a separate database.
- Although the security of both types of data is important to our business model, protecting user's private information is the highest priority. The financial data is accessed by the data harvester, a separate program gathering data from many sources on the web and introducing additional security risks. Data segregation is helps limit the scope of a potential data breach [**?**].
- The two types of data serve two separate purposes. The modules responsible for managing each are also decoupled. Thus, separation helps to enforce the principle of least concern.
- A large corpus of guides and examples on how to manage user accounts is available online. Extending any of these to include financial data might be difficult, and risks leading to bad design.

The separation of dissimilar collections of data is a practice widely adopted in industry. Criteria for assessing when this approach is appropriate have also been documented. [**?**] Based on the decision to use SQLite as our DBMS and to maximize the portability and security of the financial data, we decided to implement this decision by using two seperate databases.

4.8 The Data Layer Module (Finda)

The Finda module was designed to implement the data layer, acting as an intermediary between the data harvester/business logic and the financial data. It allows users to manage a number of databases implementing a common schema and give them access to a suite of tools for reading, writing, and removing the data stored in each. In addition to this the Finda module implements the following features:

- A system for managing user permissions to help reinforce separation of responsibilities among Thalia's other modules.
- Integrity checks to ensure the integrity of the data provided to the end user.
- A suite of administrative features to aid with managing the application back end.

Users can access these features through an outwards facing interface object, designed based on the facade design pattern [**?**].

Finda's design was modeled after object relational mappers (ORMs), libraries offered by most popular web frameworks the use of which was prohibited by the project constraints. Although the implementation of what is essentially our own ORM proved to be costly in terms of developer time, it allowed us to create a more focused module tailored to our requirements. This helped to streamline the development of other modules.

5 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.

5.1 Overview

One of the first decision we have made this term was to completely recreate our prototype of Thalia. Firstly, this radical move was the consequence of a new implementation decision (for more detail see 5.3.1). 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 the risks posed by using an Software Version Control (SVC) Host such as GitHub, we have decided to continue using it as our software development platform. The reasoning behind this builds on the argument developed in our Technical Report [**?**], which highlights that our, 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. 5.3.3 [**?**].

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

5.2 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 [**?**]. Furthermore, since no team member was the sole developer of a system component, this 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 Figure5.

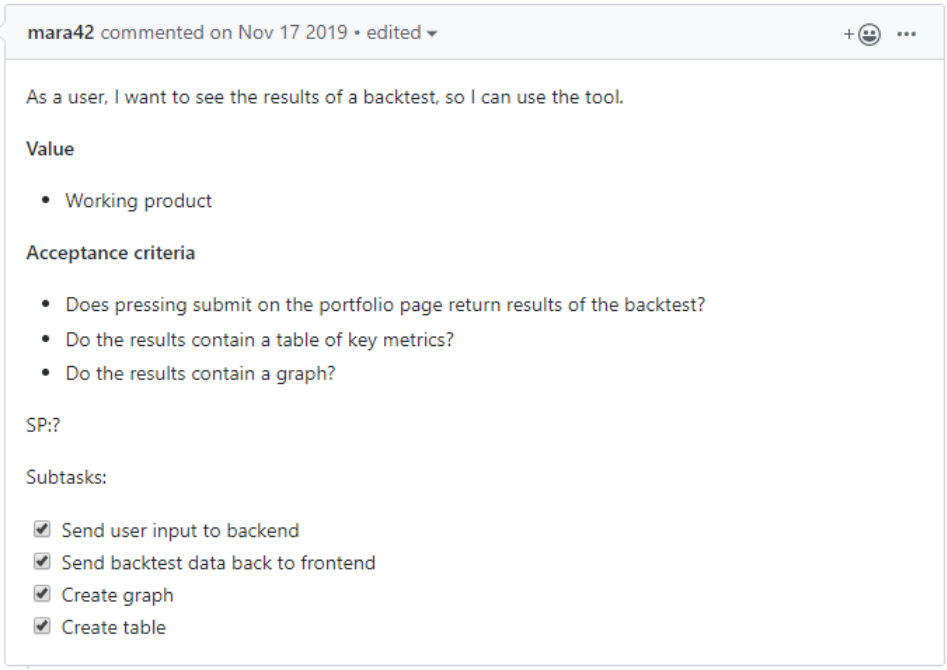


Figure 5: Ticket Example (source: https://github.com/)

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.

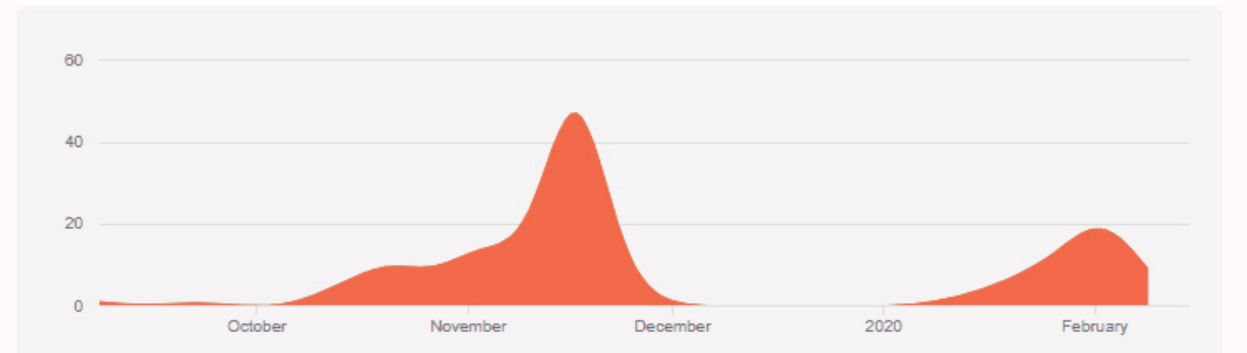


Figure 6: Contributions to the Master branch (source: https://github.com/)

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 Figure7.

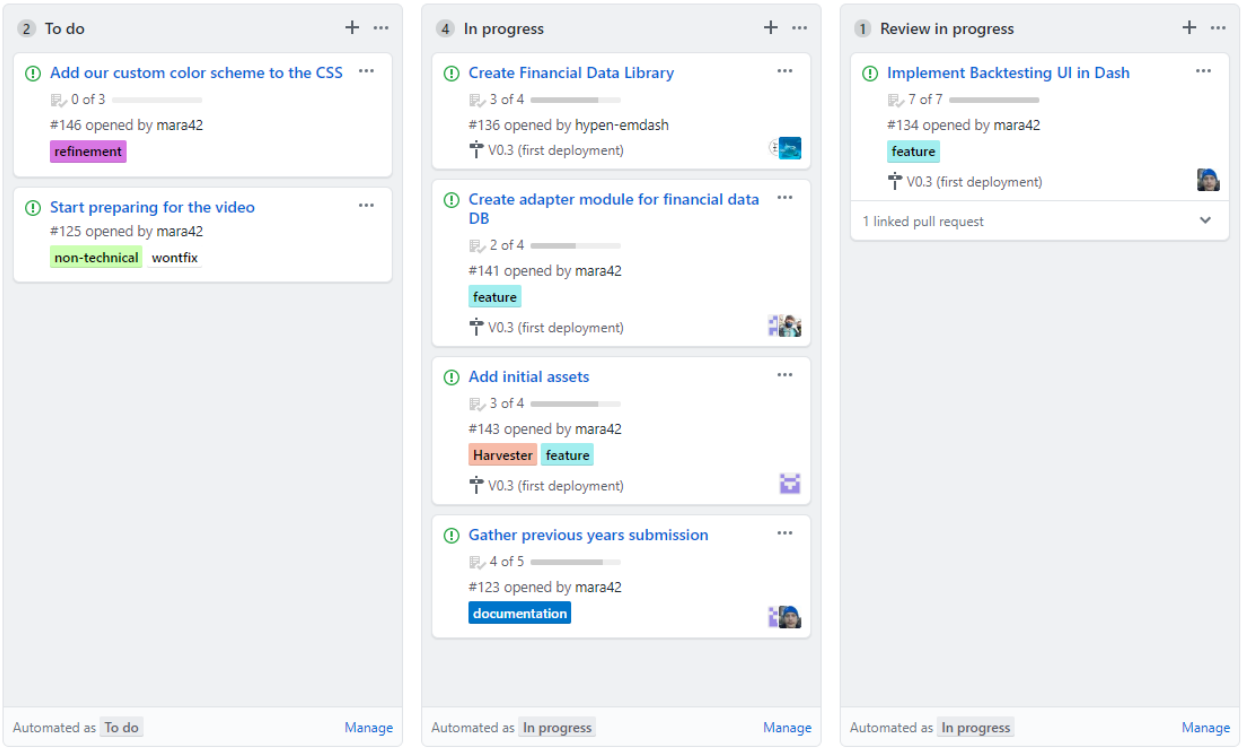


Figure 7: Scrum board - truncated

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 5.4). 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 5.4). After all checks have passed, the changes were successfully integrated into the code base.

5.3 Key Implementation Decisions

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

5.3.1 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 significant 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.

5.3.2 GUI

5.3.3 Business Logic

The business logic module sits at the very core of our application. We require it for producing a time series of a portfolio's performance as well as selected key risk metrics of a given asset allocation. This output is consumed by our web application and presented in plots and tables as described in 5.3.2.

Research into developing this module was initiated by listing our requirements for its desired behaviour. We identified that it should support:

- Specification of a portfolio as a set of pandas dataframes, the common data exchange format in our application
- Calculation of a portfolio's return over time
- Regular contributions to mimick saving
- Rebalancing strategies to reestablish the desired weighting of a portfolio
- Calculation of key metrics, including the Sharpe and Sortino Ratio, Max Drawdown, Best and Worst Year
- Collecting and reinvesting dividends for equities

Using these requirements, we struggled to find an open-source library that would handle these tasks for us. While backtesting libraries written in Python are available in abundance (e.g. PyAlgoTrade [**?**] and bt [**?**]), each of these was lacking in at least one critical aspect. None of them support specifying a portfolio using absolute or relative weights and instead seem to focus on backtesting trading strategies involving just a single asset while using technical indicators. Thus, we made the decision to develop our own library for handling the aformentioned tasks.

The result of this effort is "Anda" (short for analyse data). For each of its functions, Anda takes as input a Strategy object that specifies the entire list of parameters for a backtest, including contribution dates, a list of assets with associated price data, dividends for equities, etc. Calculations are performed on a per-metric basis by separating them into indivual functions. This approach has allowed us to tailor the entire business logic module exactly to our needs without having to produce complicated wrapper code for existing backtesting libraries.

5.3.4 Database & Finda

Another major technology decision was the choice of appropriate database management system (DBMS) for storing historical price data collected by the data harvester. Before committing to a specific technology we identified the following requirements a suitable DBMS should fulfil:

- **Schema:** The structure of our data is relatively simple, consequently Thalia does not require support for sophisticated features and data types. A suitable DBMS should be able to accommodate the database schema designed last term, with the addition of simple integrity constraints and cascade operations.
- **Support:** Ideally the DBMS should be cross platform, as this would allow us to defer commitment to a specific deployment platform until we are ready to start the CD process.
- **License and pricing:** The DBMS should be free to use and have a non-restrictive license. PERFORMANCE: The DBMS should be able to handle a high volume of concurrent reads to fulfil user requests. The data will be updated daily, meaning efficient write operations are a lower priority.
- **Usability:** As our team lacks experience in this field, a suitable DBMS should be relatively simple to learn. Ideally team members should be able to learn the basics in a single weekly sprint.
- **Security:** The DBMS should have a mature code base and be relatively secure, as access to financial data is a key component of our business model. Later it will likely also store data that is not available through public APIs, meaning potential data breaches could expose us to legal liability. [**?**]
- **Type:** Since the project constraints specify we use SQL queries, only relational DBMS supporting a version of SQL are appropriate.

MySQL, PostgreSQL, SQLite and MariaDB were subject to in depth comparison based on fulfilment of the above requirements and industry adoption [**?**] [**?**]. Our final decision was to use SQLite for the following reasons:

It is user-friendly and easy to deploy, allowing us to start continuous deployment faster. It has a small footprint and offers good performance. [**?**] Portable serverless design aids with development and testing. All team members have experience working with SQLite from previous term. This helps to reduce overhead of knowledge transfer.

The main drawbacks of using SQLite, namely scalability and performance are not a concern at this stage, as the current version of Thalia is meant to be a high quality industrial prototype, and as such will not contain the full range of financial data needed for marketability. Should SQLite prove to be inadequate in the future, we would be able to switch to a different DBMS with relatively little trouble, as the process of database migration is exceedingly well documented [**?**] [**?**]. To preempt any difficulties that might arise, the decision was made to design the data layer to easily accommodate such a migration.

5.3.5 Harvester

5.4 Integration and Deployment

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 a priority.

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:

```
"""A setuptools based setup module."""
from os import path

from setuptools import find_packages, setup

```

```
here = path.abspath(path.dirname(__file__))

install_requires = [
    "flask",
    "flask-login >= 0.5",
    "flask-migrate",
    "flask-wtf",
    "pandas",
    "dash",
]


tests_require = ["pytest", "coverage"]

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

setup(
    name="Thalia",
    version="0.2.0",
    packages=find_packages(),
    install_requires=install_requires,
    extras_require=extras_require,
)
```

Listing 3: 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 in the code snippet above. 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.

Another tool used for enforcing uniform style was the black auto-formatter for Python [**?**], which formatted the code for us upon every save if enabled, and also when commiting 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 writing the configuration file. This ensured that both flake8 and black have been run before publishing changes.

5.4.1 Continuous Integration

Many studies have investigated the positive effects of developing in a continuous integration (CI) environment [**?**], [**?**]. Regardless of the exact implementation, 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 6.

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 reveal 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, for more detail see Figure8.
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 5.4.2

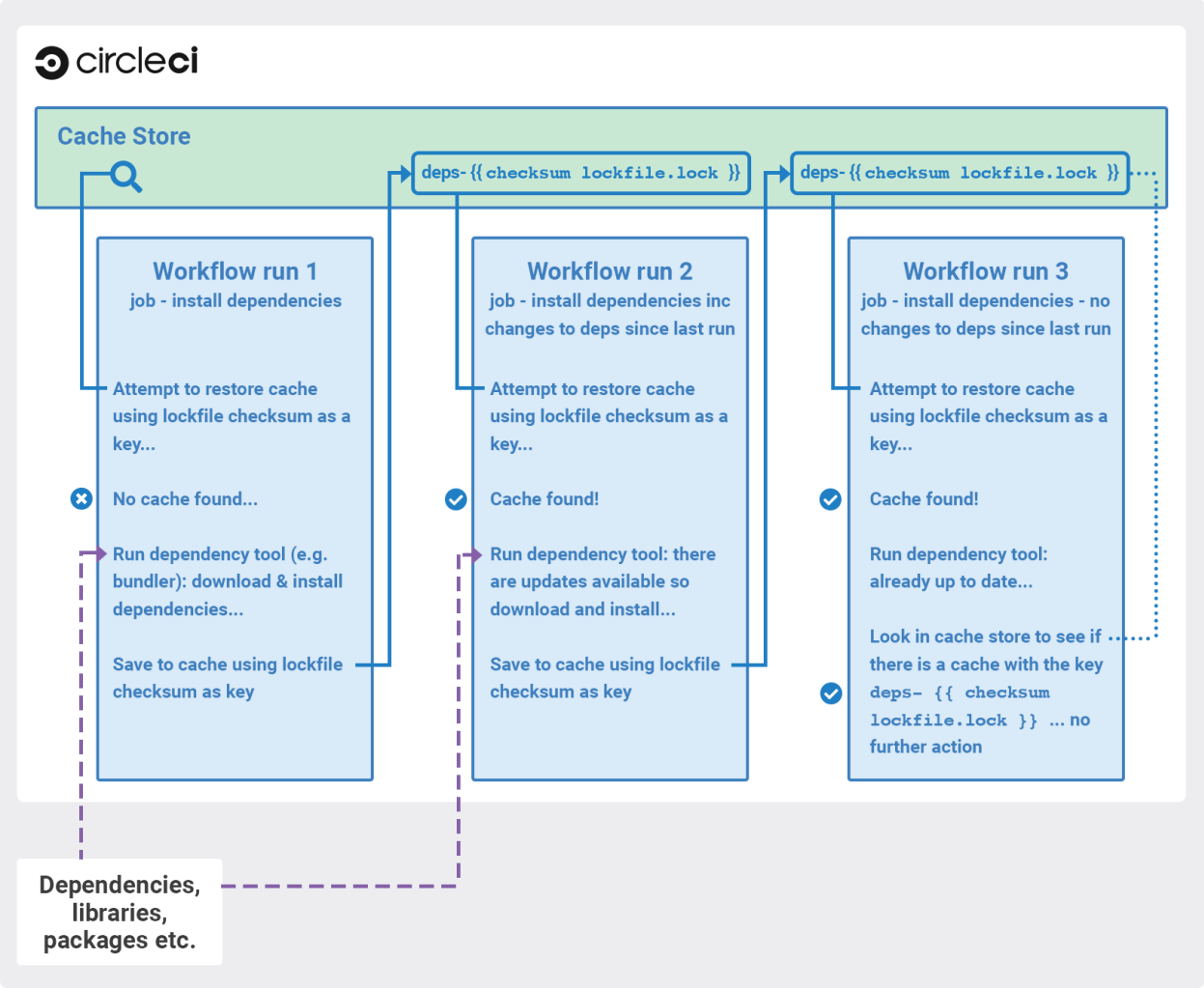


Figure 8: CircleCI on Caching (source: https://circleci.com/docs/2.0/caching/)

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.

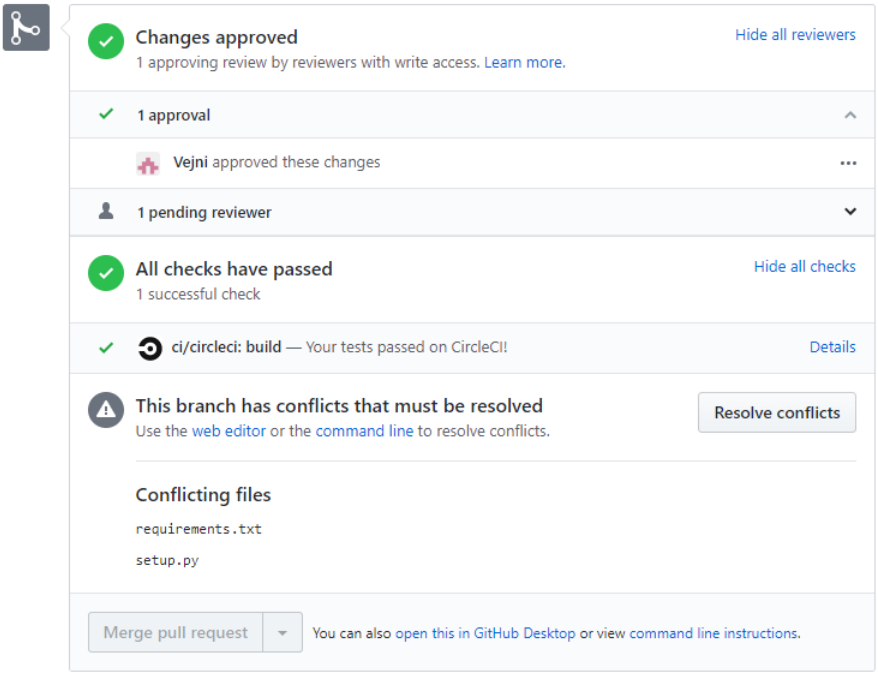


Figure 9: CircleCI on GitHub (source: https://github.com/)

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.

5.4.2 Hosting and Continuous Deployment

An overview over the literature covering reveals a plethora of different strategies for deploying and hosting web applicaitons [?]. Our decision of how to choose among them involved the following considerations:

- Price - since we have severe budget constraints, we were looking for a cheap hosting solution
- CircleCI support - The target host should be supported by CircleCI natively to ease development of a continuous deployment (CD) pipeline

The upfront cost of buying physical servers ruled it out as an option for us. Thus, we turned our attention to using a solution that involved deployment to a virtual machine in the Cloud. Many providers for such a service exist, including Amazon Web Services [?] and Microsoft Azure [?]. While these would provide us with extensive control over the hosting process, their use involves a lot of complexity that seemed unnecessary for the intial rollout of a simple application such as Thalia.

Due to native CircleCI support and a free tier service, we ended up choosing Heroku [?] as our initial hosting provider. This enables us to host our application for free in the initial stages of development while providing ample opportunity for horizontal and vertical scaling later on, if it is required.

The benefits of using continuous deployment have been well established for multiple years and involve "the ability to get faster feedback, the ability to deploy more often to keep customers satisfied, and improved quality and productivity" [?]. Using Heroku in combination with CircleCI, our CD pipeline involves the following simple steps:

1. Upon commits to the master branch on GitHub, CircleCI triggers a workflow.
2. The workflow first executes the steps listed in 5.4.1 to ensure the validity of the current codebase state.

3. If this step is successful, the master branch is pushed to a remote repository recognized by Heroku via git.
4. Heroku executes the Procfile script stored in the root of our project to start the application using a gunicorn web server [**?**].

Our deployment process is thus fully automized and immune to failing tests, as it will only complete successfully if the application is in a correct state. The full CI/CD workflow is captured by the flowchart on Figure10.

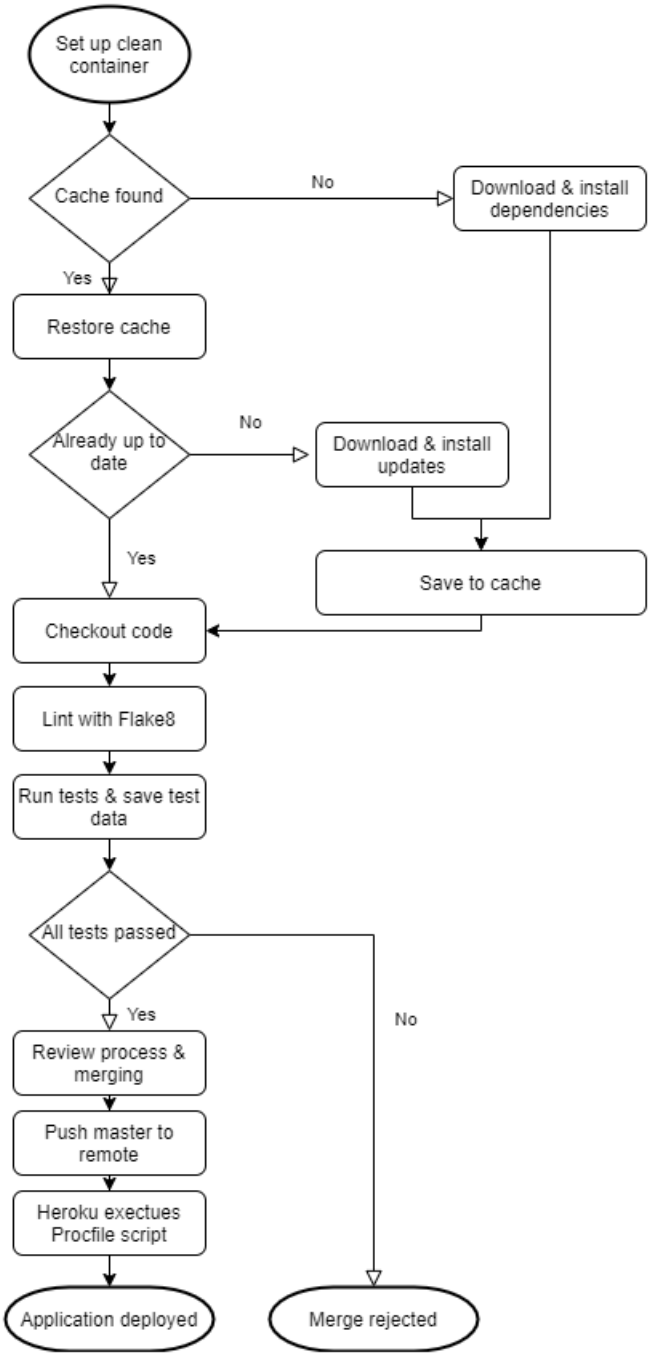


Figure 10: CI/CD workflow

6 Testing

6.1 Testing Strategy

Our initial testing strategy was devised during the first semester. Throughout development, it remained mostly unchanged. Since we opted to redesign most of our first semester's prototype from scratch we were also forced to recreate our testing suite and test data. This ultimately proved to be beneficial, as we were better able to design our tests. We also opted to spend more developer time designing our testing suite based on our experience in the first semester. We also used this opportunity to integrate several additional tools into our testing suite (last semester we only used our testing harness Pytest). In addition to a higher focus on testing and additional tooling, we also devised a testing strategy for evaluating key non-functional requirements, which will be elaborated on in [section next evaluation]. Our testing strategy consisted of three main types of the test; unit, integration and end-to-end tests. We opted to use a mix of white and black box testing techniques for the first two and conducted our end to end tests purely as black-box tests.

6.1.1 Unit Tests

All team members were required to write automated unit tests for any code they wished to merge into the project's master branch on GitHub. These were designed to test small parts of our software and its component modules and work independently of each other. As part of the code review process, team members reviewed each other's tests as well as production code. This helped us to guarantee that unit tests were exhaustive and tested the full range of input classes and boundary cases of each class and method. We also used white box testing techniques to maximize test coverage and make sure all possible paths of execution we're accounted for.

We found that this approach meshed well with our agile approach to software development, as units of work loosely corresponding to functional requirements could be added to the live version of Thalia without the risk of breaking it. This, in turn, meant that after the continuous integration process was set up, we could continually test to see that all of the components of our project worked together. Since each functional requirement had an associated suite of unit tests, we could be reasonably certain that said implementation was correct and would not act as a blocker to our progress later down the line. In addition, team members could work on Thalia's components interchangeably as per our management strategy, as any undesired side effects of one person's changes would fail the tests. Another consequence of adopting an agile approach was that we found ourselves merging code often. Unit tests helped us to identify bugs resulting from merge conflicts and in doing so help keep the live version of Thalia bug-free.

6.1.2 Integration Testing

Equally important to the unit tests were our integration tests. As with unit testing, we used a mixture of white and black box testing techniques. The majority of integration testing was done when first setting up the CI process when the individual components of Thalia first had to interact with one another. Subsequently, as we added new features, additional tests were added to the testing suite to make sure new dependencies between components we're integrated properly. Integration tests also helped us to refactor existing code to work better with other modules, as first designing breaking integration tests would help to highlight areas where interfaces would differ.

6.1.3 End-to-end Testing

Finally, upon completion of the prototype, we conducted extensive end to end testing. We used these tests to help identify system dependencies and to check that data integrity was maintained throughout the execution. They were also key in conclusively evaluating the functionality of our system, as they tested it holistically with real-world data as input.

6.2 Testing Tools

The following is the list of 3rd party tools used by our testing suite to aid with the testing of Thalia and its components:

1. **Pytest:** [citation needed] For our testing harness, we opted to use the standard python testing framework pytest. With it we could easily create a testing suite independent of production code. By using simple assert statements pytest remains easy to use, helping to reduce overhead for our team, most of whom had little to no experience writing automated tests for their code. Although simple, pytest offers several powerful features that helped us during testing:

    - Detailed introspection of failing tests helped decrease time spent debugging
    - Fixtures allowed us to automate test setup and teardown
    - Test discovery helped to keep the structure of the testing suite simple and physically separated from production code
    - Testing for expected exceptions allowed us to properly test for the unexpected flows of events

2. **Mock:** [citation needed] The Mock package allowed us to replace parts of the system with mock objects. This was useful when testing API's as it allowed us to control the input to the data harvester and allowed for repeatable execution and predictable output.

3. **Selenium:** [citation needed] Selenium is a testing tool we used to create unit and integration tests for Thalia's website. Using Firefox or Chromium in headless or full browser mode it allowed us to automate user tasks such as clicking links and input. With it we were able to test logging in, running a simulation, registering a new account, and accessing the various pages of our website.

4. **Coverage:** [citation needed] Coverage is a tool to measure code coverage in Python. It creates detailed reports on what lines of code are passed through during program execution. We used this in conjunction with Pytest to monitor what code was covered by tests to ensure our testing suite covered all possible branches of execution. In addition to validating test coverage, Coverage aided us in designing white-box tests by showing what parts of the codebase were in need of additional testing. Although not related to testing, the tool proved useful when refactoring code, as it allowed us to identify dead code.

6.3 Test Data

We selected test data for our testing suite with the following considerations in mind: For Black-Box tests, the test data should contain representatives of all major equivalence classes of the tests possible inputs. For Black-Box tests, the test data should encompass all major boundaries of equivalence classes in the accepted inputs. For White-Box testing, the data should be such that tested code is executed exhaustively.

Broadly speaking, the data we used for testing can be separated into two categories, based on how it was collected. Either we wrote code to procedurally generate the data or we used a subsection of the real world data included in the final prototype and collected from financial data API's. Each of these had its own benefits and drawbacks when being used to design tests.

6.3.1 Mock Data

In the real world, financial data, especially that relating to the prices of assets tends to be quite messy as prices tend to vary greatly from day to day [citation needed]. Part of our testing strategy was the generation mock sets of historical price data that we designed to be as clean and easy to understand as possible. This allowed us to work with neat, easily understandable datasets. For example, we created a fictitious asset whose price increased linearly over time, an asset whose price decreased and an asset whose price remained fixed. All mock data was either generated procedurally by auxiliary code we wrote ourselves or hand-coded in cases where large amounts of data were not necessary (for example when testing the functionality of the database adapter).

[image needed neat] [image needed messy]

Being able to design these custom data sets helped us to overcome the following difficulties when designing and writing tests:

- Predictability: Since complex financial equations are a key part of our service, a core requirement for our testing strategy was to be able to independently verify the results generated the Anda library. Having simple data showing, for example, a linear or quadratic increase in the daily price of an asset meant we could predict the expected results and compare them to Anda's output. With real-world data, calculating expected values by hand would have been effectively impossible.
- Interpretation of results: Components of both Thalia Web and Finda modify financial data as it passes through them. It proved difficult to see the effect processing had on real-world data. Using clean datasets meant that the effect methods had on data could be examined by a human. This was a significant aid to development and saved us considerable time in the long run.
- Designing tests: When designing black-box tests for Thalia's components, it was helpful to be able to create datasets that had specific properties for use as edge cases. A good example of this would be creating a series of prices that never decreased, as this represents an edge case for the calculation of an assets maximum draw-down [cite glossary].

6.3.2 Real-world testing data

In addition to generating our own assets for testing, we used real a limited subset of real-world data for testing across components. This is the data included in our prototype that we chose to include assets from across all supported asset classes over a significant period of time (several years as the simulation will at most span several decades). We found that this approach was easier than trying to emulate the complex fluctuations of real-world asset prices ourselves. While predicting the expected output of methods when handling real-world data was more difficult, we still found it proved useful in the following circumstances:

- Validation: Correct behaviour of system components can only be confirmed after running them on live data [citation needed]. In our case, this is especially true since the live data we work with is so complex.
- Performance: Performance of some of Thalia's features might vary based on the complexity of the input. As a result of this real data was needed to properly assess performance.
- Design: An important design consideration was how clear the UI elements, especially the ones displaying data visually (for example the price graph) looked on screens of varying form factors. Having realistic test data helped us to better assess the quality of the user experience.

6.4 Testing Results

With all unit, integration and end to end tests passing it is safe to assume that the core functionality of Thalia works as expected (short of a manageable risk of further bugs appearing, as no testing suite of this scale is perfect). Our unit tests are exhaustive, testing for the majority of possible edge cases and achieving a high (over 90 per cent) [citation needed] level of coverage for each module tested. The functionality of the final prototype was additionally tested by hand and found to be working for the included data (A range of assets from all major asset classes with over 10 years of historical data). As such it is safe to assume adding additional, structurally similar data will not break the product. Performance tests showed a potential issue with the speed of financial calculations for large portfolios, but this should only be an issue for unusually complex portfolios run over time scales of several decades [citation needed]. It is worth noting that competing software performed similarly [citation needed]. At the time of writing, we deem the testing of Thalia to be successful and to have demonstrated the suitability of our product for further development and eventually market.

7 Evaluation and further Work

7.1 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 Figure11.

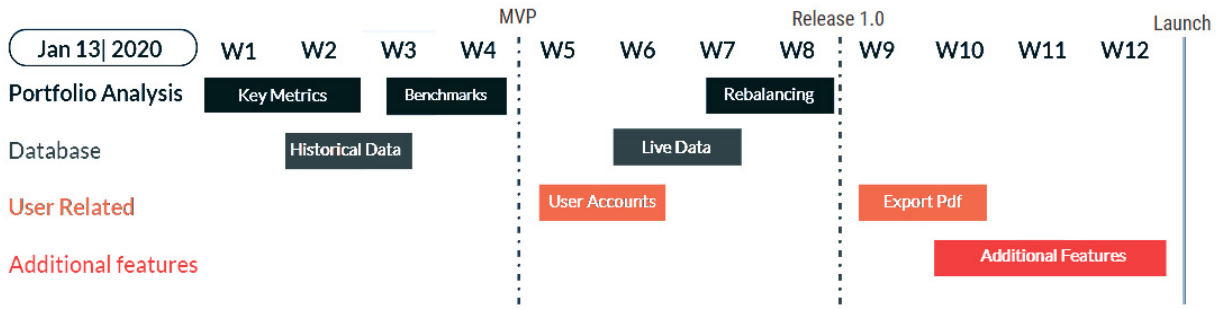


Figure 11: Technical Report: Roadmap - revisited

TODO retrospective on schedule

7.2 Feasibility Analysis

TODO

7.3 Appendix A - User Manual

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

7.4 Appendix B - Maintenance Manual

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

7.5 Appendix C - Glossary

- Portfolio - A portfolio is a grouping of financial assets such as stocks, bonds, commodities, currencies and cash equivalents, as well as their fund counterparts, including mutual, exchange-traded and closed funds. An investor's portfolio is the group of assets they have currently invested in. [https://www.investopedia.com/terms/p/portfolio.asp]
- Asset - Generally an asset that gets its value from being owned; can be traded on financial markets. Stocks, bonds, commodities, (crypto-)currencies are all types of financial assets. [https://www.investopedia.com/terms/a/asset.asp]
- Asset Class - An asset class is a grouping of investments that exhibit similar characteristics and are subject to the same laws and regulations. Asset classes are made up of instruments which often behave similarly to one another in the marketplace. [https://www.investopedia.com/terms/a/assetclasses.asp]
- Backtesting - Backtesting is the general method for seeing how well a strategy or model would have done ex-post. Backtesting assesses the viability of a trading strategy by discovering how it would play out using historical data. [https://www.investopedia.com/terms/b/backtesting.asp]
- Standard Strategy / Lazy Portfolios - A lazy portfolio is a collection of investments that require very little maintenance. [https://www.thebalance.com/how-to-build-the-best-lazy-portfolio-2466533]
- Rebalancing - Rebalancing is the process of realigning the weightings of a portfolio of assets. Rebalancing involves periodically buying or selling assets in a portfolio to maintain an original or desired level of asset allocation or risk. [https://www.investopedia.com/terms/r/rebalancing.asp]
- Key metrics - Performance measures of a portfolio that are of high interest to the majority of investors.
- Standard Deviation - The standard deviation is a statistic that measures the dispersion of a dataset relative to its mean. [https://www.investopedia.com/terms/s/standarddeviation.asp]
- Worst Year - The worst performance over any given 365 day period starting from January 1st of some year.
- Sharpe Ratio - The Sharpe ratio was developed by Nobel laureate William F. Sharpe and is used to help investors understand the return of an investment compared to its risk. [https://www.investopedia.com/terms/s/sharperatio.asp]
- Sortino Ratio - The Sortino ratio is a variation of the Sharpe ratio that differentiates harmful volatility from total overall volatility by using the asset's standard deviation of negative portfolio returns, called downside deviation, instead of the total standard deviation of portfolio returns. [https://www.investopedia.com/terms/s/sortinoratio.asp]
- Inflation - Inflation is a quantitative measure of the rate at which the average price level of a basket of selected goods and services in an economy increases over a period of time. [https://www.investopedia.com/terms/i/inflation.asp]
- Nominal Values - A value that is unadjusted for inflation.
- Real Values - A value that is adjusted for inflation.
- Equity - Equity is typically referred to as shareholder equity (also known as shareholders' equity) which represents the amount of money that would be returned to a company's shareholders if all of the assets were liquidated and all of the company's debt was paid off.

- Fixed Income - Fixed income is a type of investment security that pays investors fixed interest payments until its maturity date.
  [https://www.investopedia.com/terms/f/fixedincome.asp]
- Commodity - A commodity is a basic good used in commerce that is interchangeable with other commodities of the same type. Commodities are most often used as inputs in the production of other goods or services. The quality of a given commodity may differ slightly, but it is essentially uniform across producers.
  [https://www.investopedia.com/terms/c/commodity.asp]
- FOREX / FX - Forex (FX) is the marketplace where various national currencies are traded. The forex market is the largest, most liquid market in the world, with trillions of dollars changing hands every day.
  [https://www.investopedia.com/terms/f/forex.asp]
- Overfitting - Overfitting is a modeling error that occurs when a function is too closely fit for a limited set of data points.
  [https://www.investopedia.com/terms/o/overfitting.asp]
- Leverage - Leverage results from using borrowed capital as a funding source when investing to expand the firm's asset base and generate returns on risk capital.
  [https://www.investopedia.com/terms/l/leverage.asp]
- Technical Analysis - Technical analysis is a trading discipline employed to evaluate investments and identify trading opportunities by analyzing statistical trends gathered from trading activity, such as price movement and volume.
  [https://www.investopedia.com/terms/t/technicalanalysis.asp]

7.6 Appendix D - Glossary