# Thalia Project Report

Team Charlie

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

University of Aberdeen
Sunday 5th April, 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

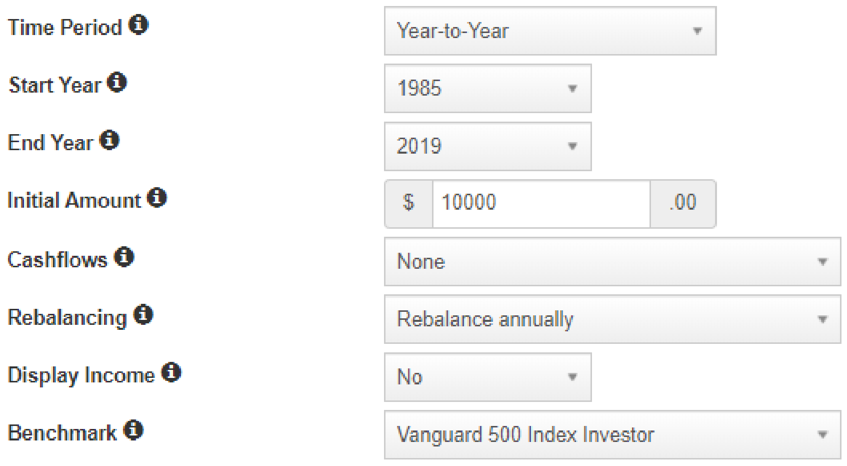
As a result of our initial competitor analysis, we may group the competing software into two categories. For a comprehensive list of available backtesters please see [?]. The first consists of various third-party trading software, such as Fidelity [?], MetaTrader [?] and NinjaTrader [?] that offer backtesting 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:



| | |
|---|---|
| Time Period | Year-to-Year |
| Start Year | 1985 |
| End Year | 2019 |
| Initial Amount | $ 10000 .00 |
| Cashflows | None |
| Rebalancing | Rebalance annually |
| Display Income | No |
| Benchmark | Vanguard 500 Index Investor |

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.

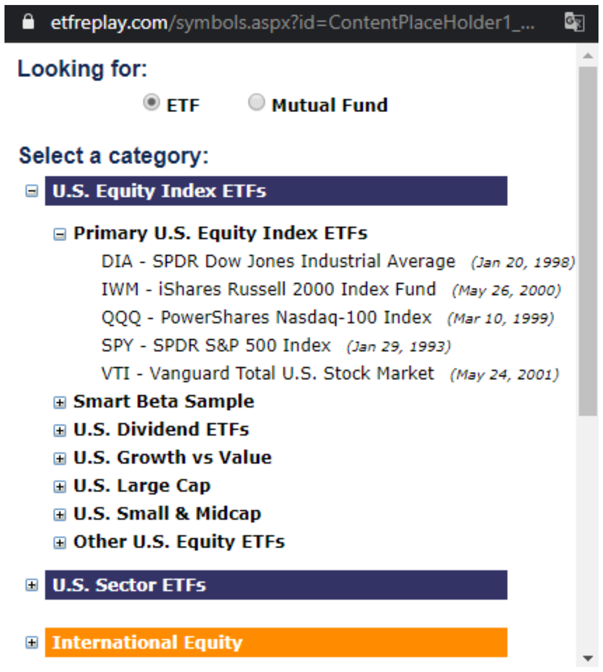

etfreplay.com/symbols.aspx?id=ContentPlaceHolder1_...
Looking for:
ETF
Mutual Fund
Select a category:
U.S. Equity Index ETFs
Primary U.S. Equity Index ETFs
DIA - SPDR Dow Jones Industrial Average (Jan 20, 1998)
IWM - iShares Russell 2000 Index Fund (May 26, 2000)
QQQ - PowerShares Nasdaq-100 Index (Mar 10, 1999)
SPY - SPDR S&P 500 Index (Jan 29, 1993)
VTI - Vanguard Total U.S. Stock Market (May 24, 2001)
Smart Beta Sample
U.S. Dividend ETFs
U.S. Growth vs Value
U.S. Large Cap
U.S. Small & Midcap
Other U.S. Equity ETFs
U.S. Sector ETFs
International Equity

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

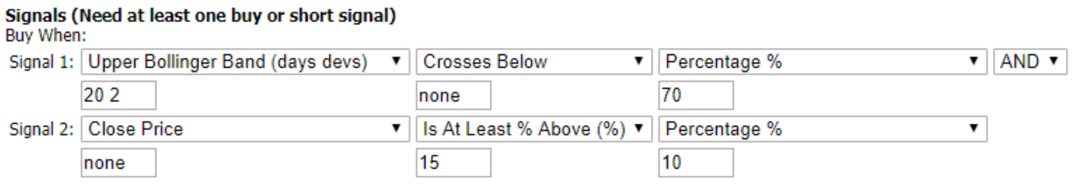
We feel this was the most intuitive to use, and will thus pursue a similar approach. The only potential issue associated with this is that it opens in a new window, which we would prefer to avoid.

## 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 [?].


Signals (Need at least one buy or short signal)
Buy When:

| | | | | |
|---|---|---|---|---|
| Signal 1: | Upper Bollinger Band (days devs) | Crosses Below | Percentage % | AND |
| | 20 2 | none | 70 | |
| Signal 2: | Close Price | Is At Least % Above (%) | Percentage % | |
| | none | 15 | 10 | |

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 Branding

## 3.1 Name and Logo

In order to find a name and a logo that would represent us, we have done a brainstorming session where we have laid out the keywords that represent our product. Out of all the words we listed, two recurring categories of keywords appeared, those were flourishing and data science. We then went on to find ways of representing those. For the flourishing part, we have decided on choosing Thalia as the name of the product. Thalia is a Greek muse often described as the flourishing, the growing one. We saw the association between our product and the Greek muse of growth as a good one given that by using our product the people who invest should also flourish and for the data science part Greeks are the modern founders of science and few other concepts relate so strongly with the idea of science.

In order to create a logo to our liking, we have asked a friend skilled in design to create that Pro Bono for us. As a result, we now have a logo that is both representative and that can be legally used.

## 3.2 Motto

Do we even have a motto?

## 3.3 Official colours

Given our team's limited experience with design, we have decided that a good way to start was to make a colour palette. After consulting some sources on Color theory [?] we have seen that we should start from choosing the base colour of our palette. In order to be up to date with the latest design trends, we have sought to use something close to the colour of the year 2020 [?]. Based on this colour we have created three colour palettes that we could use.

After some deliberation inside the team and some feedback from a few outside people, we have decided to use the third option. In the following figure, the final three colour schemes are presented with the chosen one named Thalia and third one looking left to right.

Figure 4: Color schemes [?]

03Branding/03Pictures/colors_schemes.png

# 4 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.

## 4.1 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 |

## 4.2 Non-functional Requirements

Non-functional requirements were elicited using the FURPS+ model developed by IBM to address the issues of the FURPS model, namely failure to take developer requirements into consideration [?]. The decision to use the FURPS+ model was made based on its wide adoption in the industry as well as the high volume of literature and guides for identifying key requirements available. Using this model, we identified various categories of non-functional requirements based on either user or developer needs. Elicitation of user-focused non-functional requirements was performed based on feedback from our focus group from the first semester, and the capabilities and properties of competing software (e.g. the legal requirements of providing a backtesting service). Although most requirements were identified during requirement elicitation in the first semester, several requirements and categories that we initially missed were added later during development, namely accessibility and supportability 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. Supportability:
   - Software should be well documented and easy to maintain.
   - The system should be fault-tolerant and be designed to support graceful degradation [**?**].
   - The system should be easy to install and migrate between hosting providers.
   - The system should be internationalized and be easy to extend to support new languages and currencies.
5. Implementation:
   - The system needs to work on a cloud hosting provider.
6. 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.
7. Operations:
   - An administrator on-call will be necessary for unexpected issues.
8. Packaging:
   - The product needs to work inside a Linux container (e.g. Docker).
   - All dependencies need to be installable with a single command.
9. 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.
10. 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.

## 4.3 Use Case

#### 4.3.1 Backtest Strategy

**Actors**: User

**Purpose**: Show a user how their strategy would have performed in the past.

**Overview**: The user selects they assets they want in their portfolio, along with how much to invest in each asset. They may select contribution/withdrawl and rebalancing options also. On completion, the user will see a graph plotting the value of their portfolio over time and a table of a few key metrics summarising performance.

**Preconditions**: The user must be logged in.

**Flow of Events:**

1. The user navigates to the *backtest* page of the website.
2. The system sends a list of available assets.
3. The user enters the period of time over which they wish to backtest.
4. The user selects a financial asset.
5. The user enters the proportion of their portfolio they want to invest in that asset.
6. The user submits their strategy for evaluation.
7. The system analyses the user's strategy over the specified period.
8. The user received their strategy's performance.

**Alternate Flow of Events**

- Steps 4 and 5 may repeat any number of times, allowing the user to have as large a portfolio as they like. But they must happen at least once.
- In step 6, the system may have insufficient data for the user's desired time range. If this is the case, the system should give results for the time it does have data for.
- Between steps 3 and 6, the user may optionally enter a rebalancing frequency and a contribution amount and frequency. The system should take these into account when calculating.

## 4.4 Feasibility Analysis

The feasibility of this project hinges not only on technical considerations, but must be evaluated across multiple dimensions. To conduct our analysis, we have used the TELOS methodology [?]. TELOS is an acronym for:

- Technical
- Economic
- Legal
- Operational
- Scheduling

The following sections will analyse each of these areas in detail to allow for drawing conclusions about the project feasibility.

### 4.4.1 Technical

This area concerns the question of whether there exist technologies that enable us to realise our project idea. We consider the prototype developed over the last semester sufficient evidence for the technical feasibility of our project as we have successfully developed implementations for each of our system components, albeit in stripped-down form. As a result of familiarising ourselves with the technologies we used in developing this prototype, we are confident that scaling it up to a fully functional product will pose little technical challenge to our team.

### 4.4.2 Economic

To reduced reliance on external funding for operating our business over time, license sales must outweigh costs. As part of our effort to determine economic feasibility, we have forecasted our cash flow for the year of 2020.

Figure 5: Thalia Cash Flow Forecast

03Requirements/03Pictures/cash_flow_forecast.png

This forecast makes several assumptions that we deem justifiable:

- Seed and Series A venture capital investments of a total of £75,000.
- Approximately linear growth of license sales throughout the year.
- Team members being able to sustain themselves without a salary for the four starting months.
- Being able to acquire a £20.000 business loan with less than 5% interest p.a.

These assumptions lead to our company being profitable by mid 2021. Time to profitability of around 18 months is very typical for startups, as shown e.g. by Olsen and Kolvereid [?]. While these findings are indicators of economic feasibility, we will have to constantly revise our cash flow forecast to account for changing market conditions and advances in product development.

### 4.4.3 Legal

Fundamentally, backtesting is not considered to encourage purchases or sales of assets. Our company is merely providing information to our customers and thus can not be considered to provide financial advice. Decisions drawn from the information presented by Thalia solely consists of the opinions of our customers and do not reflect the opinions of the Thalia team or any associates. A disclaimer of this form will be placed on our website and within our terms and conditions which have to be accepted by users upon sign up. Within the framework of the Financial Conduct Authority, our product is understood to provide guidance on investment decisions. Entities offering guidance ‘are responsible for the accuracy and quality of the information they provide‘ [?]. Thus, it is paramount that the information provided by our tool is accurate.

Our research has not revealed any other potential legal issues with our service. Given correctness of our calculations, we thus consider this project to be legally feasible.

### 4.4.4 Operational

After deployment of our service, we will need to continually update price data for all assets while integrating new assets to stay competitive. For a discussion of how live data is integrated into our service, please refer to the design section concerning the Data Collection module **??**. Given that these requirements are the only way in which we deviate from other web applications, we are confident that this project is feasible from an operations standpoint.

### 4.4.5 Scheduling

The Thalia Technical Report established a schedule for this sub-session. The aforementioned strict requirement of correctness of our service makes it difficult to estimate the time required for developing a system component. We are using the Agile development methodology as described in our white paper [?] as a means of adjusting for this uncertainty. Our previous schedule must then be understood to be less rigid and used instead for orientation of how to prioritise work efforts. Delivery of a correct product is thus ensured at any point. Scheduling issues are therfore less of an issue as reducing the feature specification in favour of accuracy of results is the preferred course of action.

### 4.4.6 Conclusion

The areas in the TELOS analysis yielding questions about the feasibility of this project are Economic and Legal. Accordingly, we have incorporated actions that reduce any potential negative impact of these risks into our strategy. In particular, we will be including legal disclaimers on our website and as part of our terms of service and try to acquire external funding for the initial development stage of our business as soon as possible. We believe that these discussions solidify the case for our project being feasible.

# 5 Design

## 5.1 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 (**??**). 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 [**?**].

## 5.2 Overview of System Architecture

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

Figure 6: Original Thalia System Architecture [?]

`04Design/04Pictures/architecture_layer_diagram.png`

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 **??**. The following sections will provide for a discussion of design decisions made on a layer-by-layer basis.

## 5.3 GUI Structure

The Graphical User Interface of Thalia consists of two main parts. One is a *static* website created with HTML5 [?] and stylised with the help of Bulma [?]. The second is a *dynamic* Dash [?] page, which is basically the bulk of our application. In the following paragraphs we will discuss these separately.

### 5.3.1 Thalia Web

Thalia Web consists of 5 pages, which all serve as an introduction to our application. We aimed at creating an authentic and modern looking website that attracts users. This was done by following a consistent style for the whole website, i.e. using the official colours, motto and Thalia logo, discussed in **??**.

While Thalia was not meant to be used on smaller devices due to the nature of our application, with the help of Bulma, every component of Thalia Web is responsive. Consequently, the layout of the website changes so that it fits the browser size perfectly.

Even though it is possible to navigate between pages by URLs directly, we have designed a navigation bar for this purpose. The look of this navigation bar depends on the device of the current user and whether the user has already been logged in or not. By default, an unauthenticated user will see the navigation bar visible on Figure**??**.

08Appendices/081User/081Pictures/navbar.png

Figure 7: Thalia Navigation Bar

In case Thalia is launched on a mobile phone, the navigation bar becomes a so-called "hamburger button" and dropdown menu visible on Figure**??**.

08Appendices/081User/081Pictures/navbar_hamburger.png

Figure 8: Thalia Navigation Bar - Hamburger

For the rest of this section we shall assume that the application was launched on a desktop, although the layout is identical and intuitive in both cases.

### Homepage

We shall not discuss every page separately, as they are identical. For a full walkthrough please read **??**. As an example of the design let us look at the homepage where users should arrive by default.

08Appendices/081User/081Pictures/homepage.png

Figure 9: Thalia Homepage (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/)

As users arrive at the homepage they are greeted with a short description of Thalia and the process of backtesting. We have designed the homepage such that the navigation bar and the Thalia Logo fills the screen completely. To indicate that the page does not end there, we have added a scroll down button, which with the help of a JavaScript [**?**] animation takes the user down to the bottom of the page. Here the user may encounter a small register form, or in case the user is logged in, a link to the dashboard, i.e. the main application.

08Appendices/081User/081Pictures/homepage_bottom.png

Figure 10: Thalia Homepage 2 - Top: User not logged in; Bottom: User recognised

**Log In and Sign Up Pages**

In case the User is not yet logged in, links for the ‘Log In‘ and ‘Sign Up‘ pages are visible on the navigation bar as seen on Figure**??** or Figure**??**. Both of these forms are quite common, with the login requiring:

- Username
- Password
- (Optional) Remember me

And for signing up, the fields are:

- Username

- Password
- Confirm Password

As standard, the registration fails when the user enters different values to the Password and Confirm Password fields. In this case, the user is prompted to try again.

`08Appendices/081User/081Pictures/login.png`

Figure 11: Thalia Log In Page (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/login/)

In case the user is already logged in and attempts to access these pages, they will be redirected to the homepage. Additionally, the navigation bar changes, allowing us to log out. Opting so the user finds themselves on the homepage.

### 5.3.2 The Dashboard

“Dash apps are composed of two parts. The first part is the ‘layout‘ of the app and it describes what the application looks like. The second part describes the interactivity of the application [...]” [?]. Creating the Dashboard with Dash meant using Python classes for all of the visual components of the application.

Writing code for the layout was intuitive, but cumbersome, as on top of the HTML-like structure, it also needed to be correctly indented. As such, we needed to rely on refactoring a lot, chopping the layout code into smaller, more manageable components. In our final design, we have decided to divide the Dashboard into tabs, which helped us achieve a more intuitive look and organised codebase. The layout of the dashboard can now be initialised as follows:

```
import dash_core_components as dcc
import dash_html_components as html
from . import tabs

layout = html.Div(
    html.Main(
        [
            html.Div(
                [
                    dcc.Tabs(
                        [
                            tabs.tickers(),
                            tabs.summary(),
                            tabs.metrics(),
                            tabs.returns(),
                            tabs.drawdowns(),
                            tabs.assets(),
                        ],
                        id="tabs",
                        value="tickers",
                    ),
                ],
                className="column",
            ),
        ],
        className="columns",
    ),
    className="section",
)
```

Listing 1: layout.py - Example of Dash Code

Upon arrival to the Dashboard, only the ‘Ticker Selection‘ tab is available to the user. This is where the user is expected to input their investment strategy. Although this process is fairly intuitive and is done by using dropdown menus, input fields and standard date selectors, a more thorough walkthrough can be found in **??**.

For the selection of assets we have decided to take an alternative approach visible on Figure**??**. This was done mostly because of the limitations of Dash discussed in **??**, but came with its own benefits.

08Appendices/081User/081Pictures/table.png

Figure 12: Thalia Dashboard - Assets Table

The official documentation describes the dash table as “an interactive table component designed for viewing, editing, and exploring large datasets” [?]. Working with the dash table is also quite simple, as its data can be accessed by addressing the id of the table and then the data component. For a further explanation on registering Dash Components, read **??**.

Accessing the data like this also allowed us to fulfill one of our requirements, which is to support portfolios with a large number of assets. This would have been significantly harder, if not impossible, with alternative approaches, see **??**.

To compare investment strategies, the user may add portfolios via the ‘Add Portfolio‘ button. Due to the arguments presented in **??**, we have decided to set the maximum number of portfolios to 5, compared to letting it be dynamic. In addition, it is possible to select from a set of benchmarks, also known as lazy portfolios. Having selected one, the user will find the table populated with the desired assets and proportions.

When the user is content with the input, they can start the backtesting by clicking the submit button.

If all required fields are populated, the user is taken to the summary tab. This, as well as all other tabs, are now unlocked.

### Showing the Results

Many studies have shown that one most important factor when designing a dashboard application is to show the right data using the right visualization tools [?] [?]. In our case, we have already established which metrics and plots are the most important to the users. When the user lands on the summary tab, it is these key metrics that are presented, leaving a more detailed analysis for the relevant tabs. This way the user is not overloaded with information as the results are presented.

Having decided on ‘What‘ the user should see we can now focus on the ‘How‘. In our final design, we combine data visualization techniques to make the dashboard look interesting. Key metrics are shown either in a ‘box‘, which is a core building block of a dashboard or in tables. Proportions are visualised using pie charts, whereas relative differences are shown with the help of a bar chart. Some examples can be seen on Figure**??**.

04Design/04Pictures/dashboard.png

Figure 13: The Dashboard

Among these charts the user can find one of the key components of our application, the Portfolio Growth Plot. This graph is crucial for our application, as it is a visualisation for two of our main functional requirements, i.e. showing the total returns of a portfolio over time and the comparison of strategies. Thanks to Dash this and all other graphs are fully interactive. The user may zoom in on selected areas, hover over desired data points, save the plot as an image, etc.

08Appendices/081User/081Pictures/dash_funcionalities.png

Figure 14: Dash Functionalities

Showing this in action on the Portfolio Growth Plot can be seen on Figure**??**. In this case, the user is about to enlarge the selected region to inspect the plot closer. Resetting the graph can then be done by one of the buttons visible on Figure**??**.

04Design/04Pictures/portfolio_growth.png

Figure 15: Dash Functionalities - Portfolio Growth

#### 5.3.3 Changes to the Initial Design

The design of the Dashboard was done according to the wireframes we have provided last term [**?**], also visible on Figure**??**.

04Design/04Pictures/wireframe.png

Figure 16: UI Wireframes

As we can see, we managed to encapture the overall look as we envisioned with only minor changes. The data is significantly better distributed by the use of tabs, giving a less cluttered UI. Additionally, we have decided not to implement the aside menu, partly because it was unnecessary, but mostly because it suggests that the backtest results can be changed on the go. This, as we have realised, was not feasible, due to expensive calculations in the Business Logic Layer.

One change we had to make was the use of the Dash table instead of the approach visible on Figure**??**. The benefits of this change have been illustrated further above. In addition, we have also decided to discard the sliders as a way of changing asset allocations.

For portfolio comparison, we also had to enable inputting multiple strategies. As seen on Figure**??**, this is something we have not accounted for when planning the design. The implementation of this feature led to a full refactor of the UI code, as it came with an excessive amount of code due to the arguments presented in **??**. Fortunately, this issue was addressed relatively early and resulted in the current design.

## 5.4 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]):

```python
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 2: 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:

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

Listing 3: 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.

## 5.5 Data Harvester Structure

## 5.6 Database Structure

## 5.7 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.

## 5.8 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.

# 6 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.

## 6.1 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 **??**). 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 a 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. **??** [**?**].

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

## 6.2 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 centred 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 Figure**??**.

`05Coding/05Pictures/ticket.png`

Figure 17: 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.

`05Coding/05Pictures/contributions.jpg`

Figure 18: 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 Figure**??**.

`05Coding/05Pictures/scrumboard.png`

Figure 19: 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 **??**). 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 **??**). After all checks have passed, the changes were successfully integrated into the code base.

## 6.3 Key Implementation Decisions

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

### 6.3.1 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.

For the main framework we chose Flask, a Python-based micro web framework. Flask provides functionality for building web applications, such as managing HTTP requests, using the 'requests' library, and rendering templates [?]. Flask leverages 'Jinja2' as a template engine [?].We used the 'render-template()' library to process and parse the HTML templates, for all the pages, except for the dashboard, which is described below. We used some of Flask's extensions such as Flask-Login, which provides user session management for Flask [?]. To handle web forms we used the 'flask-wtf' extension, which is a wrapper around the 'WTForms' package [?].

### 6.3.2 GUI

Dash is great for applications that require data visualisation, modelling, or analysis [?], which is precisely what is needed for our portfolio backtesting software. It allows us to create reactive single-page apps, meaning that with the use of tables, drop-down menus, etc., the app is updated instantly. This is done with the use of 'callbacks', which whenever an input changes, make an AJAX request whose output value is updated automatically [?].

A typical callback using an example from our application can be seen below:

```python
def register_add_portfolio_button(dashapp):
    dashapp.callback(
        Output("portfolios-container", "children"),
        [Input("add-portfolio-btn", "n_clicks")],
        [State("portfolios-container", "children")],
    )(add_portfolio)
```

Listing 4: setup.py - Development environment

In this code snippet, we register a listener to the button with the unique id: 'add-portfolio-btn', and its property 'n_clicks', or number of clicks. Whenever the input property changes, the corresponding function gets called automatically. This also happens upon launching the Dash page, therefore it is common to start the called function by catering to this case, and raising a 'PreventUpdate' exception if the input parameter is 'None'. An example can be seen in [?].

The called function then receives the input properties as parameters. Furthermore, we can add any number of parameters as 'State' objects, as has been done above. When the function finishes, it needs to return the same number of values as is specified in the callback. Mismatching the number of return values results in a runtime error. Whereas mismatching the input parameters yields a different function signature, and so the callback is not registered to the desired function, doing so does not raise any errors.

Fortunately, Dash offers great developer tools [?], such as the 'Callback Graph' visible on Figure**??**, which can be accessed by running the application in debug mode and activating the Dev tools. This graph came in handy often, especially due to the arguments presented in **??**.

`05Coding/05Pictures/callback_graph.png`

Figure 20: Callback Graph on Thalia

### Limitations

Although Dash is fairly intuitive and great for one-page applications, it struggles when it comes to dynamically created components. To be more precise, all callbacks need to be registered before runtime. For example, in our application the user is able to enter multiple portfolios via the 'Add Portfolio' button. Doing so creates new input fields for the portfolio which all need to be registered.

As this problem persisted with other components as well, we needed to find some solutions. Many approaches have been suggested by the Dash Community, some even by the developer team [?]. At the end, we have decided to use the most stable approach and generate all the components beforehand and hide them. This means that the relevant function now also has the responsibility of returning the visibility of the required Div. Note that this is a highly requested feature in the Dash community, and is likely to be added soon, but for now, we needed to resort to workarounds.

Furthermore, the data we show in the dashboard is greatly centered around the strategy object discussed in **??**. At the beginning of every backtesting process a new strategy object is initialised, which is a time-costly task. As this object holds most of the information our result pages require, many functions would need to access the same object.

One solution would be to have a callback for initialising the object, and then share it with all others. Although there exist ways to do so [?], we were not satisfied with any of the approaches listed. Consequently, we decided to go forward with the original approach that Dash favours, and combine more outputs in one function. Even though this introduced more complexity to our code, it made the application faster overall.

### Layout

The layout of the application, as discussed in **??**, has been created with the use of the 'Dash HTML components' library. Consequently, there was no need to write HTML for the dashboard, since the layout was composed using Python, which generates HTML content. Dash uses the Flask web framework, which makes it fairly simple to embed a Dash app at a specific route of an existing Flask app [?]. The framework also handles most of the communications between the front-end and back-end.

The User-Interface was made with the use of 'Bulma', which is an open-source CSS framework [?]. Bulma is ideal for creating responsive elements as it automatically adapts the website to different screen sizes [?]. It is also compatible with various web browsers. Another advantage of Bulma is that we could create a very modern-looking website fairly simply as the framework is doing a lot of styling for us.

It also provides pre-styled elements, components, such as the navigation bar or alerts for the users. It is fairly simple to learn in comparison to 'Bootstrap', as Bulma has easy to learn syntax, it has simple readable class names like '.button' and has a very straightforward modifiers system, for example, '.is-primary'. There are no JavaScript elements, only CSS, which makes it adaptable to frameworks such as Dash.

## 6.3.3 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 ??.

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 mimic 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 aforementioned 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 individual 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.

### 6.3.4 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 the 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 pre-empt any difficulties that might arise, the decision was made to design the data layer to easily accommodate such a migration.

### 6.3.5 Data Harvester

The Data Harvester role is to provide the application with the data it requires in order to allow the users to construct their investment portfolios. The choices we have made had to fulfil all our requirements while keeping in mind our financial and time-related limitations.

#### Chosing the APIs

We started our quest for data by trying to find APIs that could call for the main financial assets used when making an investment portfolio [?]. We have seen that there were several APIs that could be used. After a quick look at them, we have seen that there were two types of APIs. There were those that had FOREX [?] data and those designed for Stock Market Data [?]. Those two APIs that we considered for FOREX data were Fixerio and Nomics. For stock market data we have considered yfinance, Alpha Vantage and Quandl.

#### FOREX API

Between nomics [?] and fixerio [?] we have chosen nomics for the simple fact that nomics had unlimited calls for free, besides, nomics also has cryptocurrencies data. However, Fixerio is a more robust API with a larger user base and more data, that we would have used if could have afforded the investment. In the scenario where we would have had money to spend we would have used fixerio for currencies and nomics for cryptocurrencies. However, in our case we have no money so we used nomics for both currencies and cryptocurrencies. To compare the two options we have used the documentation provided by each API .

#### Stock Market Data

Out of yfinance [?], Alpha Vantage [?]and Quandl [?] we have chosen yfinance because it offered the most data and API calls for free, compared to the other two. If we had money to spend we would have used Alpha Vantage because of the larger number of available assets and the high number of requests per minute. As for Quandl we found it to be expensive and not centred on the type of data we need. We have made our choice based on the API specification given in the documentation of each API.

#### Standard Data Format

While looking for solutions that would help with solving the data format and the maintenance problem we have found pandas_datareader [?]. It is a wrapper for the biggest financial APIs. We saw that it included all of the Stock Market Data APIs named above and more. In addition, it standardizes the calling procedure for all APIs it includes and it returns the same format no matter the API you are calling. This is the only product that does this on the market so no compassion with other product can be done. In the case where we would have not used it we would have produced code that would have archived the same result. In conclusion, using pandas_datareader has saved us some valuable man-hours.

**Update the database & Redundancy & Data Interpolation**

For these requirements, we have seen that no already existing systems can do what we want. As a result, we had to write our system. In the building of the system, we have used python as a programming language. This decision has been made because the APIs chosen have been built in python. Writing the system in a different language while the APIs are in python would have created additional problems without any benefits. To manipulate the data we have used the pandas library. The persistent data required by the update mechanism is stored in CSV [?] format because of the readability and the ease of use when using pandas.

## 6.4 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:

```python
"""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 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 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.

### 6.4.1 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 **??**.

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 Figure**??**.
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 **??**

05Coding/05Pictures/cache_CI.png

Figure 21: 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.

05Coding/05Pictures/circleCI.png

Figure 22: 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.

### 6.4.2 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 **??** 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 Figure**??**.

`05Coding/05Pictures/CI_CD.png`

Figure 23: CI/CD workflow

# 7 Testing

## 7.1 Testing Strategy

Our initial testing strategy devised during the first semester remained mostly unchanged throughout development. 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 able to reassess and improve the design of our tests. We also opted to spend more developer time on testing so that we could achieve high overall test coverage to help increase code quality, reliability and maintainability [**?**]. Additionally, we used this opportunity to integrate new tools into our testing suite to aid with comprehensive testing (last semester we only used our testing harness Pytest[**?**]).

Our testing strategy so far consisted of three main types of 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. In addition to this the data harvester was designed to perform tests on the data gathered from API's at runtime, and record their results in a series of log files. The majority of our tests were automated and included in our Pytest testing suite, although a minority was also performed by hand or with the use of auxiliary code. Some examples of this would be hand testing of API constraints, as this is a task that only needed to be performed once and timing of different SQL queries to determine witch is faster.

`06Testing/06Pictures/testingSuiteStructure.png`

Figure 24: Tree of files in testing suite

### 7.1.1 Unit Testing

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 individual component modules and work independently from each other. The running of automated unit tests was performed by our CircleCI workflow. Although we are currently constrained by budget, in the future, our team plans on using the advanced metrics provided as part of CircleCI's paid plan to gain insight into how testing was performed throughout development. 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 for each software component. When designing these tests, team members also used white box testing techniques to maximize test coverage and make sure all possible paths of execution were accounted for.

In addition to uncovering bugs in our code, unit tests provided us with several other benefits. Firstly, we found that this strategy 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 our project's modules worked properly with each other. Since each functional requirement had an associated suite of unit tests, we could be reasonably certain that it was implemented correctly and would not block our progress down the line. In addition, team members could work on Thalia's components interchangeably, as per our management strategy, since any undesired side effects of one person's changes would cause the tests to fail. Finally, unit tests helped us to identify bugs resulting from merge conflicts and in doing so helped keep the live version of Thalia bug-free. This was especially important as we we're aiming to merge as quickly and as often as possible, in line with our Agile approach.

### 7.1.2 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 upon first setting up the CI process as this was when the individual components of Thalia first had to interact with one another. Subsequently, as we added new features, additional testing was performed to make sure new dependencies between components were integrated properly. Integration tests also helped us to refactor existing code to work better with other modules, since designing integration tests before writing code helped highlight areas where components' interfaces differed.

### 7.1.3 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 whether data integrity was maintained throughout execution. They were also key in conclusively evaluating the functionality of our system, as they tested it holistically with real-world data as input, which is as close to a real world environment as we could get before the Alpha release.

### 7.1.4 Data Harvester Logs

Since our data is collected from 3rd party APIs, we opted to implement a series of logs to track their behaviour at runtime. These aim to record issues with received data that stop short of causing a catastrophic failure in the Data Harvester's execution. Such a situation might occur if an API were to change its policy or experience downtime, or if there is a problem with the network on Thalia's side. In this case, we are able to look through the logs and determine what caused the error and, consequently, what parts of the Data Harvester may need modification.

The logs record where and when data retrieval failed, what data was successfully retrieved and whether the data was successfully written to the database. In addition, they perform checks at runtime to test whether the API's data is of the expected type and format.

`06Testing/06Pictures/harversterLogEx.png`

Figure 25: Example of data logged by harvester

## 7.2 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:** 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 unexpected flows of events

   `06Testing/06Pictures/failedTest.png`

   Figure 26: Example of pytest assertion introspection

2. **Mock:** 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** 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:** 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.

## 7.3 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 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 selected so 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 APIs. Each of these had its own benefits and drawbacks when being used to design tests.

### 7.3.1 Mock Data

In the real world, financial data, the prices of assets, tends to experience frequent and unpredictable fluctuations [?]. 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). 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.

### 7.3.2 Real-world testing data

In addition to generating our own assets for testing, we used a limited subset of real-world data for testing across components. 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 can be better confirmed after running them on real data [**?**]. In our case, this is especially true since the 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.

## 7.4 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 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. 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, bringing it to market. Going forward, we would like to implement a test driven development approach with the hopes of further reducing software defects and improving quality of design[**?**].

## 8 Conclusions and further work

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.

# 9 Evaluation

## 9.1 Approach

Evaluation of our product began after the team finished development and testing. As previously mentioned code was only merged after successfully passing automated tests and review by at least one other team member. We conclude that implemented features work correctly, and therefore implement their corresponding functional requirements correctly. Additional evaluation was required to assess whether or not identified non-functional requirements were fulfilled. This evaluation was performed on a requirement-by-requirement basis, and in some cases required the use of additional external tools (for example the installation of assistive technologies when evaluating accessibility requirements). In some cases, proper evaluation would have required access to resources not available to our team due to budget constraints. In these cases, the team attempted to perform evaluation as best possible and outline the difficulties experienced in this report. The team's main focus during evaluation was objectivity. To achieve this goal we relied on external tools and metrics for the evaluation of requirements where possible.

Although our evaluation process is extensive, we recognize that no amount of heuristic analysis is a perfect replacement for proper user testing, as these are two different approaches that address separate aspects of software usability and adequacy [**?**]. Additionally, some of the requirements outlined in the first semester, although important, are based on a user's subjective experience and are therefore impossible to evaluate without extensive user testing. Upon consulting with our guide and course coordinator, we were advised that user testing would be outside the scope of this course, and as such, the team decided not to conduct user testing at this stage. Extensive user testing will be performed during Thalia's Alpha and Beta releases to fully evaluate our product's usability and user experience. While conducting user testing, there is a possibility the team identifies new user personas, and consequently new requirements. As we are following an Agile approach to software development, these features can easily be added to subsequent releases of Thalia by implementing a Release on Demand process [**?**]. In the following sections, we will first discuss the results of the evaluation conducted for functional and non-functional requirements followed by a discussion of the overall success of the project.

## 9.2 Functional Requirements

As mentioned in last year's 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 Figure**??**.

05Coding/05Pictures/initial_roadmap.jpg

Figure 27: Technical Report: Roadmap - revisited

Due to our team's focus on process and code quality, the implementation of many features ended up taking longer than we initially predicted. Although the implementation of a single feature would generally be assigned as a single ticket, the writing of tests and refactoring of code to adjust for feedback would often mean tickets would take more than one week (our team's sprint length) to complete. Additionally, our team spent a lot of time implementing 'under the hood' features to improve extensibility and maintainability. Most notably, the Anda module was designed as a separate library and Finda was designed to include advanced functionality seen in other ORMs (permission management and support for managing of several databases). Although these features contribute to the quality of our product, they ended up taking developer time away from the development of functional requirements in the early stages of our project. Throughout our development process, the team would hold weekly retrospective meetings to try and better estimate velocity. These allowed us to react to this problem halfway through our allotted time. We came to the consensus decision to lock features that would be implemented and increase target weekly hours team members were expected to put into the project from 10 to 20. Additionally, we loosened some of the constraints necessary for deliverables to pass the review process and specialised the roles of team members to reduce the overhead of knowledge transfer. The result of this was that at the end of the project, all functional requirements except one were properly implemented. The only functional requirement not yet implemented at the time of writing is the ability for a user to search for assets by category when creating a portfolio. Its omission is a result of a lack of resources, architectural limitations and our internal assessment of its importance. The team assessed each functional requirement not yet implemented and determined that listing assets by category is of low importance. This decision was based on the fact that it only provides users with a slight improvement to quality of life without giving any additional insight into the viability of their portfolios. Additionally, our team determined that implementation of this feature would require architectural changes the Thalia Web and Finda modules, meaning that its implementation would represent a bad cost to value ratio when compared with other remaining features. Even without this feature, we believe this prototype to be fully featured and usable and therefore ready for the next stages of development.

## 9.3 Non-Functional Requirements

The following are the results of our evaluation of non-functional requirements identified by our team listed by category, along with some general notes on our approach to evaluating each.

### 9.3.1 Usability

Usability requirements were particularly hard to assess. This was due to their subjective nature, relying on the experiences of users. Hence we will only conclusively be able to say that these requirements are fulfilled after performing user testing.

- The product must be easily usable for users who already have some nancial investment experience.
  *Since Thalia's design and features are based on feedback from finance students and enthusiasts interviewed in the first semester, we can infer that someone of similar experience would likely be familiar with most of Thalia's terminology, features and charts.*
- The basic backtesting interface needs to look familiar to people already experienced with it.
  *The design of our interface closely follows that of our major competitor (Portfolio Visualizer), so it is safe to say that a user with moderate experience using it or other backtesting tools would have no trouble finding their way around.*
- The product must have detailed instructions on how to use its advertised functions.
  *A link to an 'About' section featuring instructions on how to use Thalia is featured in the taskbar on Thalia's landing page. Additional material is also available in the user manual* **??**.
- All major functions must be visible from the initial landing page.
  *All pages are accessible through the taskbar at the top of Thalia's landing page.*
- Must work in both desktop and mobile browsers.
  *Since Thalia's dashboard is built using the Dash platform, it automatically scales to the size of the browser window, even when accessed from a mobile browser. All other pages have also been designed to work on mobile platforms.*
- The results page should scale with mobile
  *Same as for the previous requirement.*

### 9.3.2 Reliability

- The product must have a greater than 99% uptime.
  *Thalia is hosted by Amazon AWS, a reputable hosting provider who's service level agreement (a legally binding contract) promises a monthly uptime of over 99.9%[**?**].*

- All our assets need to have up-to-date daily data where the asset is still publicly tradeable.
  *All assets included in Thalia are gathered from APIs by the Data Harvester. All API usage constraints were thoroughly researched and tested to make sure we are able to access data daily and that the data provided was updated on all trading days. This practice will continue for any API that we wish to add to Thalia.*
- All assets supported by the system must provide all publicly available historical data.
  *Both sources of data currently used by the Data Harvester (Yahoo Finance and Nomics) are publicly available and do not impose any restrictions on the use of said data.*

### 9.3.3 Performance

Performance and load times of Thalia's web page hosted on our cloud hosting provider were tested with the use of Google Lighthouse[**?**], a tool for evaluating web page quality.

- The website should load within 3 seconds on mobile.
  *Although Google Lighthouse estimated Thalia's 'time to interactive' (time needed for all interactive elements of the website to become accessible to the user) to be above the 3-second threshold, all other features of the page were loaded in this time. Additionally, Thalia scored well overall in performance.*

`07Evaluation/07Pictures/performanceLighthouse.png`

Figure 28: Results of Lighthouse Performance Tests

- Large portfolios must be supported - up to 300 different assets.
  *There is no reasonable limit on the size of the portfolio supported, and our tool has been successfully tested on portfolios of over 300 assets. Although the tool does become unwieldy when entering portfolios of that scale since the table assets are entered into becomes too large to fit on a screen without scrolling. This problem is especially pronounced on mobile.*

### 9.3.4 Supportability

- The system should be well documented and easy to maintain. *The architecture and design of the system are documented in this report. Additional documentation on the individual modules is also available on the project wiki. Additionally, the team has created a maintenance manual* **??** *explaining in detail how to maintain and extend Thalia.*
- The system should be fault-tolerant and be designed to support graceful degradation. *The system has been designed so that faults resulting from bad input and API problems do not crash the website. This means that users not directly affected can continue to run simulations.*
- Installation and migration between hosting providers should be simple. *In an appropriate environment, both Thalia and its dependencies can be installed with a single command. Installation is extensively documented in the user manual* **??**. *Since CircleCI tests all working versions of Thalia in a Docker container, our application is easy to migrate to any platform that supports Docker.*
- The system should be internationalized and be easy to extend to support new languages and currencies.
  *The current version of the Anda library supports currency conversion, as such adding new currencies is supported by our architecture. Versions of Thalia's website and dashboard in different languages can easily be implemented by hiring a translator. This is however outside the scope of our project witch is merely an high quality industrial prototype.*

### 9.3.5 Implementation

- The system needs to work on a cloud hosting provider.
  *The system is successfully deployed on Amazon AWS cloud hosting services.*

#### 9.3.6 Interfacing

- The Data Gathering Module must never use APIs stated to-be-deprecated within a month.
  *Neither Yahoo Finance nor Nomics has made any statement to this effect. As such, we can assume both APIs will remain unchanged and available in the near future. In addition to this, Nomics API's SLA (service level agreement) guarantees a high average uptime [**?**].*

- The Data Gathering Module must not exceed its contractual usage limits.
  *Fulfilled as previously stated.*

#### 9.3.7 Operations

- An administrator on-call will be necessary for unexpected issues.
  *This requirement would require the hiring of additional staff and is therefore outside the scope of this project. We have however included a form for submitting complaints on the Thalia website.*

## 9.4 Packaging

- The product needs to work inside a Linux container (e.g. Docker).
  *As previously mentioned, our CircleCI process' tests are all run in a Docker container. This means that throughout development all working versions of Thalia have been tested and are therefore proved to be working inside one.*

- All dependencies need to be installable with a single command.
  *Dependencies necessary for deployment are installed automatically when Thalia is installed with Pip. Testing and extra dependencies can be installed additionally, but are not necessary for running Thalia.*

#### 9.4.1 Legal

Proper evaluation of the fulfilment of legal requirements would require consulting a legal professional, as none of the team have an in-depth understanding of British law. This is again outside the scope of this project. Nevertheless, the team has attempted to comply with the identified legal requirements to the best of our abilities so as to try and minimise potential future changes. The steps we have taken to fulfil these requirements are outlined below.

- All user testing must be done with ethical approval from the University.
  *We were advised by the course coordinator that user testing is not necessary for this stage of our project. Additionally, we were advised that as long as no personal information is published, user testing could proceed without such approval. Therefore this is a false requirement.*

- UI must display a clear legal disclaimer about the service not providing financial advice.
  *A disclaimer is clearly displayed on Thalia's website. Since our tool doesn't propose a specific course of action to our users, it is likely it would constitute at most financial guidance and not financial advice[**?**]. In addition, our disclaimer is based on disclaimers found in the terms of service of competing software [**?**] and should, therefore, require little modification after review by a legal professional.*

- All third-party code should allow for commercial use without requiring source disclosure (e.g. no GPL-3).
  *With the use of the Pip package manager, we were able to confirm that the majority of third-party code does not require us to disclose our source code. The remaining package was identified to be nomics-python, which is used to access the Nomics API and is distributed under the MIT license, meaning it also allows commercial use without source code disclosure.*

07Evaluation/07Pictures/licensingPip.png

Figure 29: Licensing of 3rd party packages

- User data handling should comply with GDPR.
  *The team has done its best to familiarize itself and comply with GDPR and related legislation[**?**].*

- Provided services should not constitute nancial advice under UK law to avoid being subject to nancial advice legislation and potential liability.
  *Thalia is merely a tool for running simulations, and as such we at no point advises users on how to invest their real-life assets. This is also stated explicitly in our legal disclaimer.*

### 9.4.2 Accessibility

- Display items should be clearly labelled.
  *All UI elements are clearly labelled based on their purpose. All elements of the dashboard have been given descriptive names.*

- UI should scale to accommodate dierent screen sizes and aspect ratios.
  *All pages on Thalia's website have been designed to scale to different-sized desktop and mobile displays. Dash apps automatically scale and rearrange elements to fit the display, meaning the main dashboard scales as well.*

- UI elements and text superimposed over one another should have high contrast in their colours.
  *When selecting Thalia's colour scheme, care was taken to allow for such contrast. On the dashboard, foreground elements and text are rendered in dark blue and black, while background elements are white or bone. A similar colour scheme is used for the other pages on Thalia's website, with foreground and background colours inverted.*

- UI should allow for the use of assistive technologies to accommodate for individuals with accessibility issues.
  *Selection of assistive technologies to test our website was made based on guidelines set out by the UK government identifying the 5 most commonly used assistive technologies [**?**]. As these work by magnifying parts of the screen and reading out website text, they did not encounter any issues on our website. This means that Thalia in its current form reaches the high standard set for UK government institutions and associated businesses.*

## 9.5 Evaluation Results

The prototype of Thalia we have delivered along with this report fulfils the vast majority of identified functional requirements. We have also provided adequate reasoning for why we decided not to implement the remaining functional requirement. When possible, evaluation was also performed for each non-functional requirement. We believe that we have shown why these too are adequately fulfilled. In addition, our team has focused on the quality of our process throughout Thalia's development, and as a result of this has been able to deliver high-quality code that has been extensively reviewed and tested. Based on these factors we evaluate that our project has been successful, and we have delivered a high quality, functioning piece of software that fits our identified business needs and market niche.

## 9.6 Appendix A - User Manual

### 9.6.1 Launching Thalia

As Thalia is a web application, it is accessible in a web browser via the url: http://ec2-54-211-238-18.compute-1.amazonaws.com/.

`08Appendices/081User/081Pictures/thalia_domain.png`

Figure 30: Thalia Web

Navigation within the application can be achieved mainly by the navigation bar, visible on Figure**??**.

08Appendices/081User/081Pictures/navbar.png

Figure 31: Thalia Navigation Bar

In case Thalia is launched on a different device, such as mobile or in a smaller window, the layout changes to fit to the screen. In this case the navigation bar becomes a so called "hamburger button" and dropdown menu visible on Figure**??**.

`08Appendices/081User/081Pictures/navbar_hamburger.png`

Figure 32: Thalia Navigation Bar - Hamburger

For the rest of this manual we shall assume that the application was launched on a computer, although the layout is identical and intuitive in both cases.

### 9.6.2 Homepage

By default the user ends up on the homepage, although some other pages are accessible as well given the correct url. The Homepage is visible on Figure**??**.

08Appendices/081User/081Pictures/homepage.png

Figure 33: Thalia Homepage (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/)

The purpose of the homepage is to provide a cover to our application. As users arrive at the homepage, they are greeted with a short description of what the software actually is. As we do not wish to have users jumping blindly into the process of portfolio analysis, following that we provide some basic information on the backtesting process. A link at the end of the description then leads to the about page, where users can read more on the topic.

Scrolling further down the user may encounter a small register form, or in case the user is logged in, a link to the dashboard, i.e. the main application.

08Appendices/081User/081Pictures/homepage_bottom.png

Figure 34: Thalia Homepage 2 - Top: User not logged in; Bottom: User recognised

### 9.6.3 About Page

The about page of our application is to provide a more detailed description of the problem domain, and is available at http://ec2-54-211-238-18.compute-1.amazonaws.com/about/. The user can arrive at this page either by directly typing in the url, clicking on the learn more option on the Homepage, or by navigating here using the navigation bar.

Our About Page looks as follows:

`08Appendices/081User/081Pictures/about.png`

Figure 35: Thalia About Page (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/about/)

### 9.6.4 Log In and Sign Up Pages

TODO if we introduce extra fields or regex for pw.

In case the User is not yet logged in, links for the Log In and Sign Up pages are visible on the navigation bar as seen on Figure**??** or Figure**??**. In addition, they are directly available at http://ec2-54-211-238-18.compute-1.amazonaws.com/login/ and http://ec2-54-211-238-18.compute-1.amazonaws.com/register/. Both of these forms are quite common, with the login requiring:

- Username
- Password
- (Optional) Remember me

And for signing up, the fields are:

- Username
- Password
- Confirm Password

As standard, the registration fails when the user enters different values to the Password and Confirm Password fields. In this case the user is prompted to try again.

`08Appendices/081User/081Pictures/login.png`

Figure 36: Thalia Log In Page (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/login/)

In case the user is already logged in and attempts to access these pages, he or she gets redirected to the homepage. In addition the navigation bar changes, allowing to log out as visible on Figure**??**. Opting to log out the user finds themselves at the homepage.

### 9.6.5 Report Issues Page

The Report Issues Page is available at http://ec2-54-211-238-18.compute-1.amazonaws.com/issues/ or via opening the dropdown menu on the navbar and clicking the link. This short form is for users to provide feedback, report possible issues or request features.

`08Appendices/081User/081Pictures/issues.png`

Figure 37: Thalia Report Issues Page (source: http://ec2-54-211-238-18.compute-1.amazonaws.com/issues/)

### 9.6.6 TODO PAGE

### 9.6.7 Dashboard

The Dashboard, our main application, is available only if the user is logged in at http://ec2-54-211-238-18.compute-1.amazonaws.com/dashboard/ or via the link on the navigation bar. This web page is divided into tabs, which are:

- Ticker Selector
- Summary
- Metrics
- Returns
- Drawdowns
- Assets

At first only the Ticker Selector Page is accessible for the user. This is because the following tabs show only the output of backtesting a or multiple portfolios, and as a consequence are, for the time being, empty.

08Appendices/081User/081Pictures/disabled_tabs.png

Figure 38: Thalia Dashboard - Disabled Tabs

Let us now consider each tab individually.

**Ticker Selector**

Here the user is required to input their backtesting strategy. As Thalia supports testing multiple portfolios at once (up to 5 currently) there are input fields that are portfolio specific and that are not. The latter consists only of:

- Start Date: The day the investment is made.
- End Date: The last day of the investment, defaulted to the present date.
- Initial Amount: Initial balance, default currency is dollars.

The first two of these are standard date-selectors, but also allow for typing the date directly. Next, the user enters the data specific to the current strategy.

- Portfolio Name: Defaulted to Portfolio 1, Portfolio 2, etc.
- Contribution Amount: The amount that should be regularly invested into the portfolio.
- Contribution Frequency: Specify how regularly should these contributions be.
- Rebalancing Frequency: Specify how often rebalancing should happen.
- Investment Proportions : Discussed below.

The last one is the most crucial information in a strategy. First the user selects the desired asset from the dropdown menu. The menu item also allows for typing in order to find an asset faster. The selected item is then added to the table below.

`08Appendices/081User/081Pictures/table.png`

Figure 39: Thalia Dashboard - Asset Table (source: TODO)

The user then specifies the proportion of the investment, this is required to be a numeric value, representing the relative weight of the asset. For example, given Asset A with relative weight 1, and Asset B with relative weight 2, then they 33% of the investment comes from Asset A and 66% from Asset B. In case the user is content with the portfolio, they can either add another strategy via the "Add Portfolio" button, or click "Submit" to see the results. In case the user has already entered the maximum number of portfolios, i.e. 5, the button is disabled and the only possibility is to submit.

In many cases the user may want to compare their portfolio to a benchmark, that is to a small set of predetermined portfolios. These can be selected from the "Lazy" dropdown at each portfolio, which then populates the table with the desired proportions.

TODO User uploaded data

**Summary**

If all required fields are populated, the user is taken to the summary tab. This as well as all other tabs are now unlocked. At this point the user is shown one of the key components of our application, i.e. the portfolio growth graph. Thanks to Dash this, and all other graphs are fully interactive. The user may zoom in on selected areas, hover over desired data points, save plot as image, etc.

`08Appendices/081User/081Pictures/dash_funcionalities.png`

Figure 40: Dash Functionalities

The plot seen at the top ofFigure**??** shows the total return of each portfolio. Users will typically use this graph for comparison. In addition, for each portfolio the user has given as input, the following is shown:

- Selected Key Metrics: Final Balance, Best Year, Worst Year, etc.
- Pie Chart: Proportion of each asset.

- Bar Chart: Shows how the total return changed over the years relatively to the last.

`08Appendices/081User/081Pictures/summary.png`

Figure 41: Dash Functionalities

**Metrics**

**Returns**

**Drawdowns**

**Assets**

## 9.7 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.

## 9.8 Appendix C - Problem Domain 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]

## 9.9 Appendix D - Project Terminology

- Anda - A library that calculates a portfolio's historical performance and associated risk metrics. The name has been chosen as the abbreviation of ‘analyse data‘.
- Data Harvester - Component of our system that fetches data from third party financial data APIs and writes it to the Thalia database. Retrieves both historical and live price data as well as dividends data for equities.
- Finda - An interface for our databases. Provides functionality for registering, creating and connecting to databases as well as reading from and writing data to them. The name has been chosen as the abbreviation of ‘find data‘.
- Thalia Web - Shorthand for the Thalia Web Server providing the service our customers interact with.