

Thalia Project Report

Team Charlie

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



University of Aberdeen
Sunday 19th April, 2020

Our thanks go out to Dr. Beacham and Dr. Compatangelo for their continued support and guidance, to our logo designer and video assistant Marton Kottmayer and the folks at Stackoverflow for assistance in any technical matters.

Contents

1	Introduction	5
1.1	Project Overview	5
1.2	Motivation	5
1.3	Project Management Strategy	5
1.4	Budget	6
2	Background and Competitors	7
2.1	User Types	7
2.2	Competitor Analysis	8
2.2.1	Portfolio Visualizer	8
2.2.2	Other Competing Software	9
2.2.3	Summary	11
3	Branding	11
3.1	Name and Logo	11
3.2	Motto	12
3.3	Official colours	12
4	Requirements	13
4.1	Functional Requirements	13
4.2	Non-functional Requirements	15
4.3	Use Case	17
4.3.1	Backtest an Investing Strategy	17
4.4	Feasibility Analysis	17
4.4.1	Technical	18
4.4.2	Economic	18
4.4.3	Legal	19
4.4.4	Operational	19
4.4.5	Scheduling	19
4.4.6	Conclusion	19
5	Design	19
5.1	General Design Decisions	19
5.2	Overview of System Architecture	19
5.3	GUI Structure	20
5.3.1	Thalia Web	20
5.3.2	The Dashboard	23
5.3.3	Changes to the Initial Design	26
5.4	Business Logic Structure	27
5.5	Data Harvester Structure	28
5.5.1	Choosing APIs	28
5.6	Database Structure	29
5.6.1	Data Segregation	29
5.6.2	The Data Layer Module - Finda	30
6	Coding and Integration	30
6.1	Overview	30
6.2	Planning	31
6.3	Key Implementation Decisions	33
6.3.1	Web Framework	33
6.3.2	GUI	34
6.3.3	Business Logic	36
6.3.4	Database and Finda	36

6.3.5	Data Harvester	37
6.3.6	Continuous Updating Mechanism and Maintenance	37
6.3.7	Redundancy	38
6.3.8	Standard Data Format	39
6.3.9	Data Interpolation	41
6.3.10	Security	41
6.4	Integration and Deployment	42
6.4.1	Continuous Integration	43
6.4.2	Hosting and Continuous Deployment	45
7	Testing	48
7.1	Testing Strategy	48
7.1.1	Unit Testing	48
7.1.2	Integration Testing	49
7.1.3	End-to-end Testing	49
7.1.4	Data Harvester Logs	49
7.2	Testing Tools	50
7.3	Test Data	51
7.3.1	Mock Data	51
7.3.2	Real-world testing data	52
7.4	Testing Results	52
8	Evaluation	52
8.1	Approach	52
8.2	Functional Requirements	53
8.3	Non-Functional Requirements	54
8.3.1	Usability	54
8.3.2	Reliability	54
8.3.3	Performance	55
8.3.4	Supportability	55
8.3.5	Implementation	56
8.3.6	Interfacing	56
8.3.7	Operations	56
8.4	Packaging	56
8.4.1	Legal	56
8.4.2	Accessibility	57
8.5	Evaluation Results	58
9	Conclusion and Further Work	58
9.1	Conclusion	58
9.2	Further Work	58
9.2.1	Payment Service Integration	58
9.2.2	Implementation of a Scripting Language	58
9.2.3	Additional Risk Metrics	58
9.2.4	Expanding Asset Selection	58
9.2.5	Equity Dividends	59
10	Appendices	60
11	Appendix A - User Manual	60
11.1	Launching Thalia	60
11.2	Browsing Thalia	60
11.2.1	Homepage	61
11.2.2	About Page	62
11.2.3	Log In and Sign Up Pages	63

11.2.4 Contact Us Page	64
11.2.5 About Us Page	65
11.2.6 My Portfolios Page	65
11.3 Running a Backtest	66
11.3.1 Dashboard	66
11.3.2 Ticker Selector	66
11.4 Viewing the Results	68
11.4.1 Summary	68
11.4.2 Metrics	69
11.4.3 Returns	69
11.4.4 Drawdowns	70
11.4.5 Assets	71
11.4.6 Overfitting	71
12 Appendix B - Maintenance Manual	73
12.1 Prerequisites	73
12.2 Installation	73
12.2.1 Registering a Database	73
12.3 Running the application in a Development Environment	74
12.3.1 Running the application in a Production Environment	74
12.4 Test instructions	74
12.5 Source Code Files	74
12.6 Anda	74
12.7 Data Harvester	74
12.8 Finda	75
12.9 Thalia	76
12.9.1 Dashboard	76
12.9.2 Flask	77
12.9.3 Selenium Tests	77
12.10Miscellaneous	78
13 Appendix C - Problem Domain Glossary	79
14 Appendix D - Project Terminology	81
15 Appendix E - UML Diagrams	82
15.1 Anda	82
15.2 Data Harvester	83
15.3 Finda	84
15.4 Thalia Web	85
16 Appendix F - User Types Diagrams	86
References	94

1 Introduction

1.1 Project Overview

The aim of this project was to create a portfolio backtesting tool that allows the creation of custom portfolios and the measurement of their performance on historical data. The user would be able to pick from a variety of assets, including various Equities, Fixed Income Securities, Currencies, Commodities, and Cryptocurrencies. Risk metrics and performance are then calculated, visualized and displayed to the user for analysis. Thalia is our attempt at creating such a tool.

1.2 Motivation

Our target audience is the growing market of retail investment, which consists of individuals who, due to preference or financial limitations, choose to eschew professional money management solutions such as the hiring of financial advisers, and instead take on the task of assessing the viability of their investment strategies themselves. Retail investors tend to be non-professionals who invest comparatively small amounts and have few resources at their disposal. To fill our identified market niche and cater to these investors, we aimed to design a tool that would be accessible, affordable and contain data on a wide variety of financial assets for investors to choose from as this is something current solutions fail to provide [1].

1.3 Project Management Strategy

The creators of Thalia are:

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

Our goal for this semester's project was the creation of a high quality industrial prototype of the Thalia backtesting tool, based on our identified requirements and optional features [2]. 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 for the discussion of ideas and the outlining of requirements were held weekly. Later, during the implementation stage, the team held internal meetings at least once per week for the discussion of the tool's design as well as several joint programming sessions.

As in the past [2], our workflow was centred around the GitHub platform and the tools it provides. Furthermore, we continued to follow the Egalitarian Team structure, as this worked well during the first term of the course. In the previous term's Technical Report, we 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, this term we concluded that in many situations these metrics had failed to accurately measure and predict the size of a task. Based on this observation, we decided to abolish this practice for the rest of the development process.

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 resulted in better-quality of the codebase and faster team communication and decision making.

1.4 Budget

As previously discussed, we did not have any budget restrictions other than time. The team discussed personal expectations in advance and agreed to allocate a minimum of 10 hours per week to development. We estimate the value of our product's 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 [3] (taking into account that we have no industry experience, i.e. that the value of our work is towards the lower end of the range provided).

2 Background and Competitors

The following sections provide a discussion of our defined target audience (identified with the help of User Types) and an in-depth analysis of competing backtesting tools.

2.1 User Types

During the process of identifying user types, we mainly relied on the methodology described in [4]. In addition, the advice given by [5] and [6] proved invaluable. We began by gathering as much data as possible about the categories of people interested in backtesting and consequently in investing. The next step was to create several skeleton personas [7] and to validate their legitimacy through discussion with the team and cross-referencing with information gathered about various types of investors. Based on this, we decided that the creation of four personas would cover the vast majority of our target demographics. These four personas created are described here with the full charts attached to this document in appendix 16.

- **Addriana:** Addriana is a 56-year old flower shop owner who is trying to invest in order to gain a comfortable retirement. She has previously used the services of a financial advisor but was left disappointed. As a result, she wants to have more control and more hands-on knowledge of the investments she is making.
- **Allison:** Allison is a finance student in her 20s who wants to use the knowledge she acquired from university in a way that would allow her to both make a profit and gain insights to help in her career.
- **Sam:** Sam is an accomplished middle-aged man who wishes to know more about investing and the benefits and disadvantages that come with it.
- **Timmy:** Timmy is a high-school student who is interested in short term investments. He would describe himself as being part of the new wave of cryptocurrency enthusiasts.

With the personas created, we conceived of a number of scenarios where backtesting could be of value. We then put each persona in turn through them. This process helped us to extract the empathy maps [8] of our personas. We then validated our empathy maps with our team and with outsiders to provide a third party perspective. The validation with people from outside the team proved to be crucial in adjusting for internal biases. Based on the user personas and their empathy maps, we then began looking at what types of metrics we could best use to describe them. We identified the critical ones to be the knowledge on the topic of finance and the amount of funds available for investing. Another metric that was found to be important was that of the various types of investing timeframes (differentiating between users aiming at short or long term investments). Based on these metrics, we identified the following four user types:

1. **Financial Proficiency and Large amounts of Capital:** This person already knows the benefits of backtesting and is only looking for a tool that can speed up the process of constructing an investment portfolio. The resources they have meant that they would likely be able to afford our product with ease. This type of user would likely engage in both long and short term investments.
2. **High Financial Knowledge and Little Capital:** This is most likely someone young who has a keen interest in finance or is studying it at a university. Price could be a deterrent for them if their main aim is to conduct short term investing. If they would be doing long term investing, however, the price would likely not pose a problem as they would consider it a necessary part of their initial investment.
3. **Little Financial Knowledge and Large amounts of Capital:** This is a person who wants to switch from having a savings account or employing a financial advisor. They are aware of the learning curve and need additional information before they are willing to commit to any major decisions. They will be happy to pay for our product in order to gain access to our data, analysis and general advice.
4. **Little Financial Knowledge and Little Capital:** This last category is made up of individuals who lack confidence in their ability to invest successfully. As a result, they do not want to risk large sums

of money until they are sure that they can make the right decisions. There is a good chance that they will purchase our product to form a clearer idea of what investing as an activity consists of. After gaining confidence and experience they will likely increase the funds they are comfortable investing.

2.2 Competitor Analysis

As a result of our initial competitor analysis, we can group competing software into two categories. For a comprehensive list of available backtesters, see [9]. The first category is that of full-suited trading software, such as Fidelity Active Trader Pro [10], MetaTrader4 [11] and NinjaTrader [12], which offer backtesting as an additional feature. Services in this category focus on features such as the buying and selling of financial assets, prediction of future prices and storage and management of a user's capital. 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 within the scope of our prototype.

The second category consists of software that does not offer trading as a service and instead focuses solely on backtesting. Thus, for now, we shall only consider the feature set provided by the second category, as it is directly comparable to the service we are aiming to provide. In the following paragraphs, we will consider some of the design decisions made by competing software, together with a brief analysis and conclusion on each feature.

2.2.1 Portfolio Visualizer

Our main competitor of the second category is an online backtesting tool named Portfolio Visualizer [13]. During the inception phase of development, we relied heavily on this website for requirement analysis and elicitation. 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 (i.e. testing an allocation of assets with a fixed initial investment over a fixed time period), we see that in addition, users are able to perform the following set of actions:

- Set the endpoints of the investment period, up to months.
- Set the initial investment.
- Specify a regular cash flow and its frequency.
- Select a rebalancing strategy.

- Select a benchmark strategy for comparison.
- Compare multiple strategies at the same time.
- Adjust for inflation.
- Select from a set of lazy portfolios (benchmarks).
- Calculate additional metrics.
- Export the results to PDF, Excel, or save a link.

At first, it may seem as if Portfolio Visualizer meets all the requirements needed for a backtesting tool. Indeed, our main criticism is with respect to the UI, responsiveness and user experience, findings based on our initial user testing.

The UI design is simplistic and has a non-commercial, bare-bones look. Although this may be 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.

The website also supports 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 displays a heavy US bias and hence the selection of asset classes is limited.

2.2.2 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 summarised by the following question:

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 the like. Within each of these classes there are many more options to choose from. Many backtesters opted for a search bar, which is a sensible approach. Nevertheless, many portfolio items are named similarly, and for a new user, it is a high barrier to entry as they might not know what is available.

Another option is what ETFReplay [14] has implemented, which is a tree-like selection form, shown below.



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

However, pursuing this form of gathering input from the user involves multiple repetitive actions of expanding the tree structure, which becomes tedious once one is more familiar with the tool. Thus, we have chosen to pursue the search bar approach.

Tiles

It may be worth briefly discussing one example of a backtester with a good design. We found the simplistic and tiled design of Backtest Curvo [15] 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 5.

Simple Scripting Language

In our whitepaper, we briefly discussed implementing a simple scripting language which would allow users to simulate dynamic trading strategies [1]. As mentioned in our white paper, this would not be in the scope of the course, but for inspiration, we can have a look at the approach of Stockbacktest [16].

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, it demonstrates that this feature is possible to implement. We believe that after nailing down precisely how 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 previously, this is something we would like to implement in the future.

2.2.3 Summary

After testing competitor software during the inception phase of development, we were left with three key observations. These were the following:

- Transparency: Many of the testers' web pages did not give the impression of being operated by a reputable business. In the worst case, they gave the impression of being nontransparent 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 they believe 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 providing a fresh design. 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 otherwise 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

We have started the process of creating a brand for our product by looking at books and articles on branding. The two most useful resources we have seen were [17] and [18]. Based on the knowledge acquired, we have decided that our brand needs a name, a logo and a motto.

3.1 Name and Logo

In order to find a name and a logo that would represent us, we have conducted several brainstorming sessions where we have laid out the keywords that represent our product. Out of all the words we listed, two recurring categories appeared, which were 'flourishing' and 'data'. We then went on to find ways of representing those. At the end, we have decided on choosing 'Thalia' as the name of the product, which is the name of a Greek muse often described as 'the growing one'. We saw the association between our product and the Greek muse of growth as beneficial.

To create a logo to our liking, we have asked a young designer willing to create it Pro Bono for us. After a few iterations, we had two logos made from copyright free images, the smaller of which can be seen on Figure4.



Figure 4: Thalia Logo - Small

3.2 Motto

Based on our previous findings in 2.2.3, we have decided that our motto should be something that would reassure our customers that they have made the right choice. As a result, we have come up with a simple motto: ‘Thalia, the reliable Backtester’.

3.3 Official colours

The idea of defining a set of official colours came from homogeneity. We strove for a uniform design both for our product and this report, so we believed it was a good way of ensuring this. Given our team’s limited experience with design, we have decided to start by 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 [19]. Based on this colour, we have created three colour palettes by using a triadic colour scheme [20].

In the following figure, the final three colour schemes are presented. After some deliberation within the team and some feedback from a few outside people, we have decided to use the option labelled ‘Thalia’.



Figure 5: Color Schemes

4 Requirements

During the inception phase, we classified our requirements using the established FURPS+ model [21]. Since identifying the requirements listed in our initial project report [2], we have identified additional requirements that Thalia should fulfill 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)

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 previously popular FURPS model, namely failure to take developer requirements into consideration [22]. The decision to use the FURPS+ model was made based on its wide adoption in industry as well as the high volume of literature and guides available for the identification of key requirements. 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 new non-functional 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 the process.
- 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 whenever the asset is still publicly traded.
- 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 with up to 300 different assets in one portfolio.

4. Supportability

- Software should be well documented and easy to maintain.
- The system should be fault-tolerant and be designed to support graceful degradation [23].
- 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.
- The 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 for individuals with accessibility issues.

4.3 Use Case

4.3.1 Backtest an Investing Strategy

Actors: User

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

Overview: The user selects the assets they want in their portfolio, along with how much to invest in each asset. They may also select contribution, withdrawal and rebalancing options. 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 *dashboard* 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 receives 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. However, 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 [24]. 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 of 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 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 reduce reliance on external funding for the operation of 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 6: Thalia Cash Flow Forecast

Thalia Cash Flow Forecast												
Starting cash on hand	£ 10,000.00											
	Starting date	Jan 2020										
Cash on hand (beginning of month)	Cash minimum balance alert £ 2,000.00											
	January 2020	February 2020	March 2020	April 2020	May 2020	June 2020	July 2020	August 2020	September 2020	October 2020	November 2020	December 2020
Cash on hand (beginning of month)	£ 10,000.00	£ 37,690.00	£ 34,530.00	£ 32,120.00	£ 29,960.00	£ 19,050.00	£ 8,890.00	£ 48,980.00	£ 37,820.00	£ 29,910.00	£ 21,750.00	£ 14,340.00
Total												
Cash Receipts												
License sales	£ -	£ 500.00	£ 1,000.00	£ 1,500.00	£ 3,000.00	£ 4,000.00	£ 4,000.00	£ 3,000.00	£ 6,000.00	£ 6,000.00	£ 6,500.00	£ 7,500.00
Business Loan	£ 20,000.00	£ -	£ -	£ -	£ -	£ -	£ -	£ -	£ -	£ -	£ -	£ 20,000.00
Venture Capital	£ 25,000.00	£ -	£ -	£ -	£ -	£ -	£ 50,000.00	£ -	£ -	£ -	£ -	£ 75,000.00
Total Cash Receipts	£ 45,000.00	£ 500.00	£ 1,000.00	£ 1,500.00	£ 3,000.00	£ 4,000.00	£ 54,000.00	£ 3,000.00	£ 6,000.00	£ 6,000.00	£ 6,500.00	£ 7,500.00
Total Cash Available	£ 55,000.00	£ 38,190.00	£ 35,530.00	£ 33,620.00	£ 32,960.00	£ 23,050.00	£ 62,890.00	£ 51,980.00	£ 43,820.00	£ 35,910.00	£ 28,250.00	£ 21,840.00
Cash Paid Out												
Advertising	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 2,400.00
Contract work	£ -	£ 250.00	£ -	£ 250.00	£ -	£ 250.00	£ -	£ 250.00	£ -	£ 250.00	£ -	£ 250.00
Materials and supplies (in COGS)	£ 14,000.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 100.00	£ 15,100.00
Office expenses	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 150.00	£ 1,800.00
Rent or lease	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 1,000.00	£ 12,000.00
Loan Repayments	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 1,300.00	£ 15,600.00
Repairs and maintenance	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 250.00	£ 3,000.00
Licenses	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 210.00	£ 2,520.00
Utilities	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 200.00	£ 2,400.00
Wages (less emp. credits)	£ -	£ -	£ -	£ -	£ 10,500.00	£ 10,500.00	£ 10,500.00	£ 10,500.00	£ 10,500.00	£ 10,500.00	£ 10,500.00	£ 84,000.00
Total Cash Paid Out	£ 17,310.00	£ 3,660.00	£ 3,410.00	£ 3,660.00	£ 13,910.00	£ 14,160.00	£ 13,910.00	£ 14,160.00	£ 13,910.00	£ 14,160.00	£ 13,910.00	£ 14,160.00
Cash on hand (end of month)	£ 37,690.00	£ 34,530.00	£ 32,120.00	£ 29,960.00	£ 19,050.00	£ 8,890.00	£ 48,980.00	£ 37,820.00	£ 29,910.00	£ 21,750.00	£ 14,340.00	£ 7,680.00

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. A time to profitability of around 18 months is typical for startups, as shown e.g. by Olsen and Kolvareid [25]. 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 the purchase or sale of assets. Our company is merely providing information to customers and thus can not be considered to provide financial advice. Decisions drawn from the information presented by Thalia are based solely on the opinions of customers and do not therefore reflect the opinions of the development team or any associated party. A disclaimer of this form will be placed on our website and within our terms and conditions, which will need 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” [26]. 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 6.3.5. 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 [1] 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 therefore 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 the pandas Dataframe [27] object as the common data format for both data exchange and calculations. Pandas provides us with data structures “that cohere with the rest of the scientific Python stack” [28], such as the module NumPy, which we are using to calculate historical returns and risk metrics (5.4). Additionally, it is supported natively by many third party APIs and can be used to easily read data from SQL databases. For additional information, please consult the official pandas documentation [27].

5.2 Overview of System Architecture

Before discussing any specific component and architectural layer in depth, it is worth revisiting our original architecture [2].

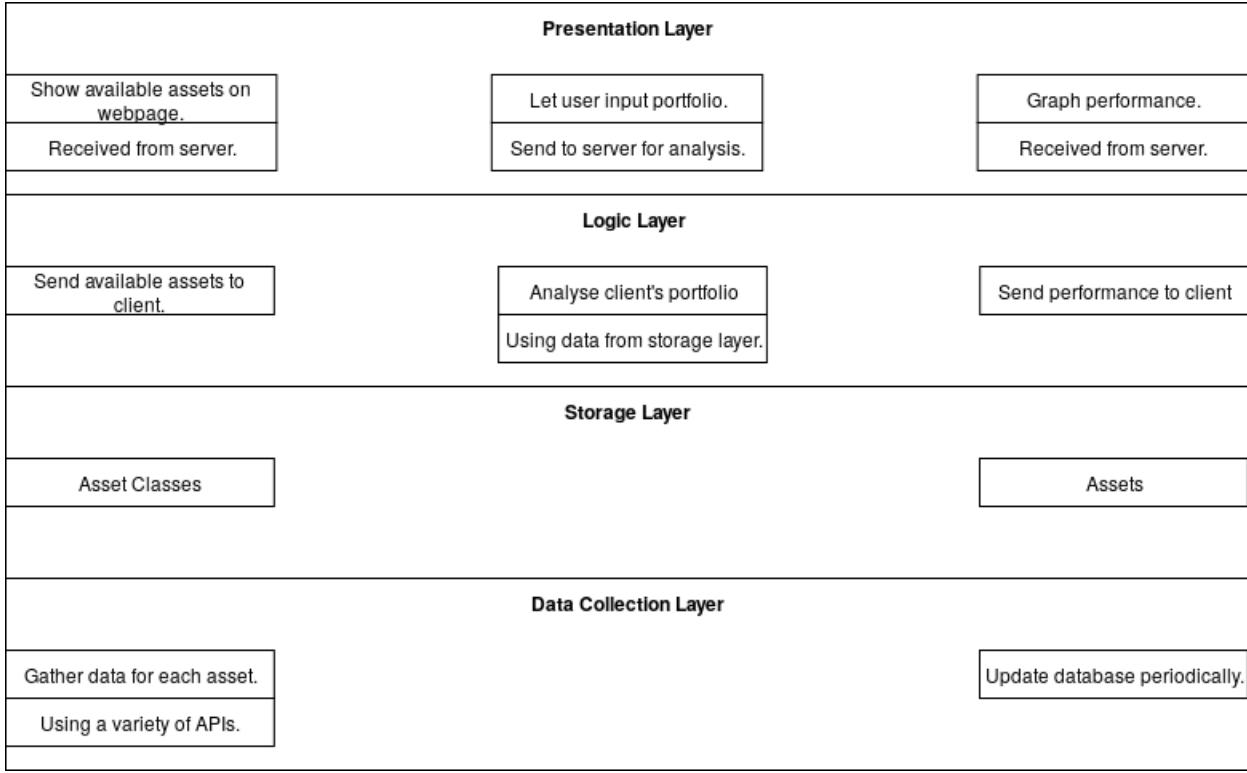


Figure 7: Original Thalia System Architecture [2]

The above schematic illustrates how we modified a typical Three Layer architecture 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 is 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 access 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 [2].

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 5.6. 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 [29] and stylised with the help of Bulma [30]. The second is a *dynamic* Dash [31] 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 would appear attractive to users. This was done by following a consistent style over the whole website, i.e. using the official colours, motto and Thalia logo, as discussed in 3.

Due to the nature of the service it provides, the Thalia website was not meant for use on smaller devices. However, 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 Figure8.



Figure 8: 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 Figure9.

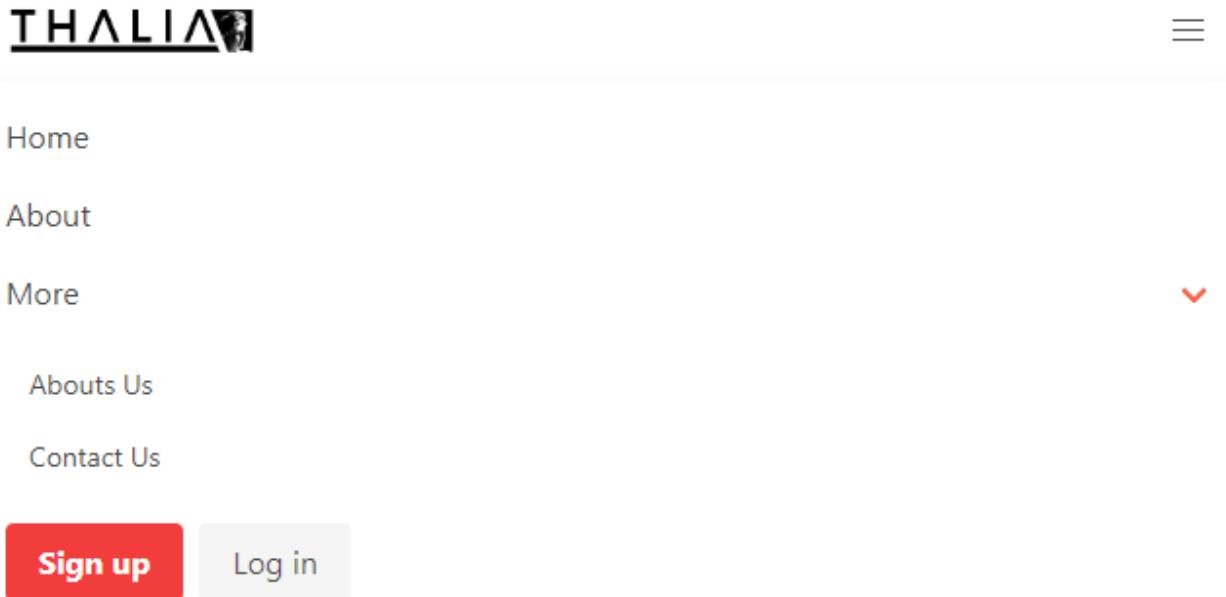


Figure 9: Thalia Navigation Bar - Hamburger

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 the mobile case.

Homepage

We shall not discuss every page separately, as they share a similar design. For a full walkthrough, please refer to 11. As an example of our design choices, let us look at the homepage where users should arrive by default.

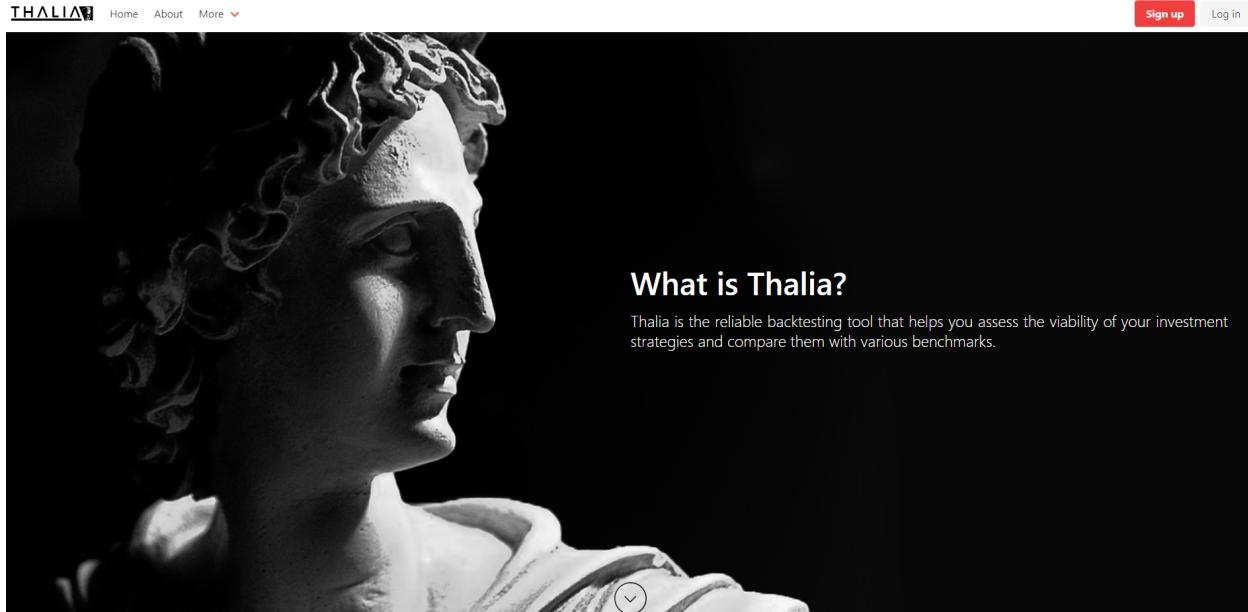


Figure 10: Thalia Homepage (source: <http://thaliabacktest.xyz/>)

As users arrive on 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 [32] 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.

A screenshot of the Thalia registration page. The top section is titled "Register" and contains three input fields: "Username", "Password", and "Confirm Password". Below these is a red "Register" button. To the right of the "Register" button are links for "Already have an account?" and "Login". A horizontal line separates this from the bottom section. The bottom section is titled "Tweets" and shows a tweet from the user @Thalia99627941. The tweet reads: "Help pass the time during lockdown by perfecting your investment strategy with the #Thalia backtesting tool!". It includes a timestamp of "Apr 5, 2020" and a "View on Twitter" link. On the left side of the bottom section, there is a "Still with us?" message encouraging users to register or log in, and a "Already signed in?" message with a note about following on social media.

Figure 11: 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 Figure8 or Figure9. 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.

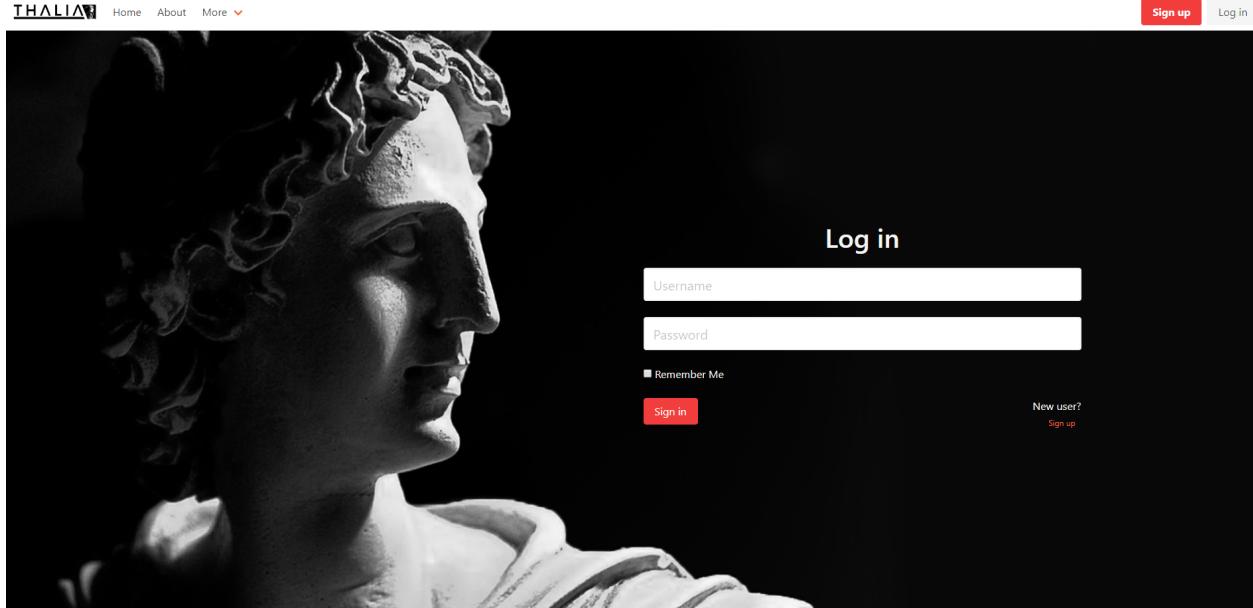


Figure 12: Thalia Log In Page (source: <http://thaliabacktest.xyz>)

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.

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 [...]” [33]. 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. Consequently, 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 a better organised codebase. The layout of the dashboard can now be initialised as follows:

```

1 import dash_core_components as dcc
2 import dash_html_components as html
3 from . import tabs
4
5 layout = html.Div(
6     html.Main(
7         [
8             html.Div(
9                 [
10                dcc.Tabs(
11                    [
12                        tabs.tickers(),
13                        tabs.summary(),
14                        tabs.metrics(),
15                        tabs.returns(),
16                        tabs.drawdowns(),
17                        tabs.assets(),
18                    ],
19                    id="tabs",
20                    value="tickers",
21                ),
22                ],
23                className="column",
24            ),
25            ],
26            className="columns",
27        ),
28        className="section",
29    )

```

Listing 1: layout.py - Example of Dash Code

Upon arriving at 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 done with the use of dropdown menus, input fields and standard date selectors, a more thorough walkthrough can be found in 11.

For the selection of assets we have decided to take an alternative approach visible on Figure13. This was done mostly because of the limitations of Dash discussed in 6.3.2, but came with its own benefits.

Ticker:			
JPY			x ▾
AssetTicker	Name	Allocation	
SPY		1	
VTSAX		2	
JPY		1	

Figure 13: Thalia Dashboard - Assets Table

The official documentation describes the dash table as “an interactive table component designed for viewing, editing, and exploring large datasets” [34]. 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, see 6.3.2.

Accessing the data like this also allowed us to fulfil 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 6.3.2.

To compare investment strategies, the user may add portfolios via the ‘Add Portfolio’ button. Due to the arguments presented in 6.3.2, we have decided to set the maximum number of portfolios to 5, as opposed to having it scale dynamically. 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 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, is now unlocked.

Showing the Results

Many studies have shown that one of the most important factors when designing a dashboard application is to show the right data using the right visualization tools [35] [36]. In our case, we have already established which metrics and plots are 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 soon 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 for users. 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 Figure14.



Figure 14: 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 serves as 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, all graphs are fully interactive. The user may zoom in on selected areas, hover over desired data points, save the plot as an image, etc.



Figure 15: Dash Functionalities

The result of performing one such action on the Portfolio Growth Plot can be seen on Figure16. 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 Figure15.

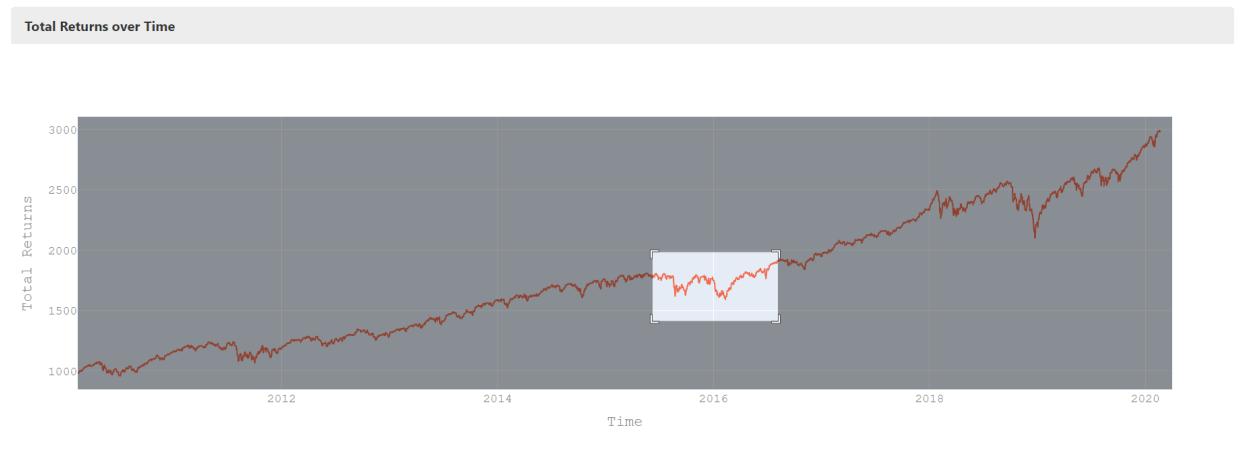


Figure 16: Dash Functionalities - Portfolio Growth

5.3.3 Changes to the Initial Design

The design of the Dashboard was done according to the wireframes we have provided last term [2], also visible on Figure17.

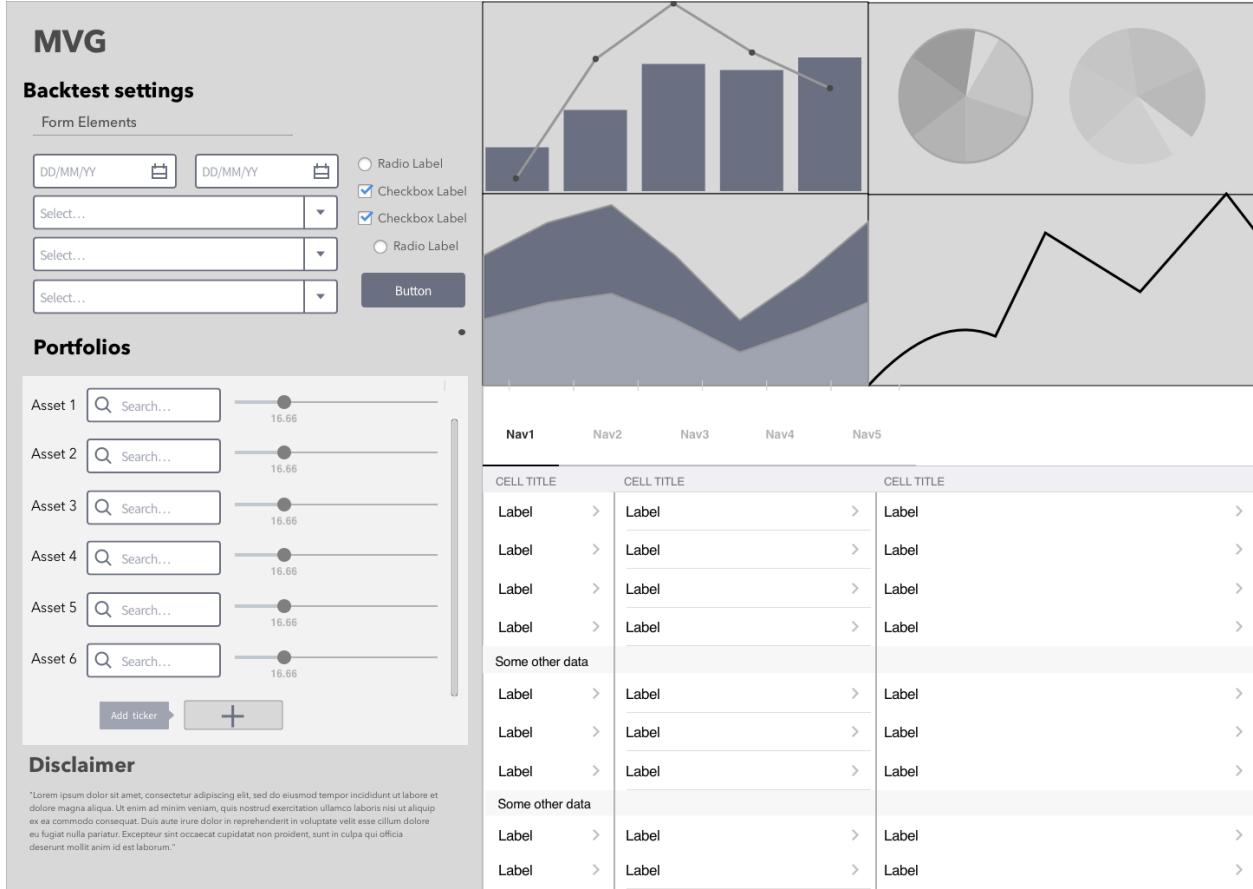


Figure 17: UI Wireframes

As we can see, we managed to encapture the overall look we envisioned with only minor changes. The data is significantly better distributed by the use of tabs, yielding 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 come to realise, is 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 17. 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 17, this is something we have not accounted for when planning the design. The implementation of this feature led to a full refactoring of the UI code, as it came with an excessive amount of code due to the arguments presented in 6.3.2. 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 14) that performs the necessary calculations. Anda is decoupled from both the presentation and database layers by relying on external providers for any price data and the specification of a strategy. This decision has allowed us to use alternative sources of price data. Another feature resulting from this is allowing users to input their own price data for assets not supported by Thalia. This data is uploaded as a CSV file. 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 required. Currently supported metrics include ‘Total Return’, ‘Max Drawdown’, ‘Best / Worst Year’, and the ‘Sharpe’ and ‘Sortino’ Ratios (please refer to 13 for definitions). 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):

```

1 import pandas as pd
2
3 class Strategy:
4     def __init__(self,
5                  start_date: date,
6                  end_date: date,
7                  starting_balance: Decimal,
8                  assets: [Asset],
9                  contribution_dates, # implements __contains__ for date
10                 contribution_amount: Decimal,
11                 rebalancing_dates, # implements __contains__ for date
12                 ):
13         self.dates = pd.date_range(start_date, end_date, freq="D")
14         self.starting_balance = starting_balance
15         self.assets = assets
16         self.contribution_dates = contribution_dates
17         self.contribution_amount = contribution_amount
18         self.rebalancing_dates = rebalancing_dates
19

```

Listing 2: Strategy Object Specification

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). An overview over the class relationships can be found in 15.

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

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

Listing 3: Example Function Signature

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” [37]. As rounding errors and imprecision are unacceptable for our application, using the Decimal type will allow us to reliably compute figures for prices, risk metrics, etc.

Finally, we have chosen NumPy [38] for performing numerical calculations as this allows for highly optimized computation through the use of vectorized operations.

5.5 Data Harvester Structure

The Data Harvester’s role is to provide the application with the data it requires in order to allow 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 constraints.

5.5.1 Choosing APIs

We started our quest for data by trying to find APIs that could be called to retrieve price data for the main financial assets we decided the Thalia prototype should support. We have seen that there were several APIs

that could be used. After an in depth comparison, we observed that there were two types of APIs: the class of APIs providing data for the foreign currency exchange market (FOREX) and cryptocurrencies, and the class providing data for global markets and exchanges. The two APIs that we considered for gathering FOREX data were Fixer [39] and Nomics [40]. For stock market data, we considered using yfinance [41], Alpha Vantage [42] and Quandl [43]. Our comparison of these was based on the documentation published by each API.

FOREX API

Between Nomics and Fixer [39], we chose to use Nomics, as it provided unlimited free API calls. Although Fixer is a more robust API with a larger user base and a wider range of available data, it requires a subscription fee to access many of its more desirable features. Additionally, Nomics also serves data for many popular cryptocurrencies, whereas Fixer only records the prices of Bitcoin.

Stock Market Data

Out of yfinance, Alpha Vantage [42] and Quandl, we have chosen yfinance. Compared to the other options, yfinance provides both more data and more free API calls. If we had had the budget, we would have opted to use Alpha Vantage due to the superior offering of assets and requests offered by its paid plan. As for Quandl, we found it to be expensive and not focused on the types of asset we were looking for.

Standard Data Format

Pandas_datareader [44] is a wrapper our team decided to use for accessing APIs. It standardizes the calling procedure and return format for all supported APIs. As it is the only product of its class on the market, no comparison of alternatives was done. Opting to use this library benefited our team by saving development time and allowing us to create cleaner code. Additionally, adding support for additional APIs supported by pandas_datareader would be a near trivial task.

Database Updating Mechanism, Redundancy and 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 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 a CSV [45] format to allow it to be both human-readable and easy to use with the pandas library.

5.6 Database Structure

5.6.1 Data Segregation

The decision was made early on to horizontally partition the data stored 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 during the last semester was the sale of collected financial data as a packaged product. This process would be trivial 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 helps limit the scope of a potential data breach [46].

- 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 [47]. Based on the decision to use SQLite as our DBMS and to maximize the portability and security of the financial data, we decided to use two separate databases.

5.6.2 The Data Layer Module - Finda

The Finda (for a definition see 14) module was designed to implement the database 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 correctness of the data provided to the end user.
- A suite of administrative features to aid with managing the application on the back end.

Users can access these features through an outwards facing virtual interface, designed based on the facade design pattern [48].

Finda's design was modeled after object relational mappers (ORMs), which are libraries offered by most popular web frameworks. 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. The use of raw SQL queries was also an explicit requirement for this project.

UML diagrams for the Finda module can be found in 15.

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 decisions made this term was to completely recreate our prototype of Thalia. This radical move was the consequence of a new implementation decision (for more detail see 6.3.1) and designed to give us the opportunity to refine the structure and quality of our code. The team determined this decision to be reasonable, as prototypes are meant to be disposable and are designed only to answer key questions about the system [49].

Despite the risks posed by the use of 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 last semester's Technical Report [2], which highlights that our Data Processing Module is a separate component of our system that none of the other components are 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 risk of accidental exposure of sensitive data.

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, which are as follows:

- Python is a high level programming language allowing us to focus on the application.
- A standard choice for prototyping.
- Provides superb third-party libraries and frameworks free of charge.
- Easy to integrate with other languages if we were to choose to do so at a later stage of 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 [50].

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 decided to adopt the Egalitarian Team structure to make use of the flexibility offered by it. This meant that individual team members would not have fixed responsibilities, allowing anybody to work on each of the systems components. This decision also eased the code review process, as there were no internal power dynamics that might have adversely affected the error-correcting mechanism [51]. Furthermore, since no team member was the sole developer of a system component, this meant review comments could be directed at the code and not an individual author [52].

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 usually allocated by the team at the weekly meetings, but occasionally they were chosen by the person proposing the feature or change. Our goal with this approach was to divide larger jobs into smaller tasks, so units of work could be completed in a single sprint in line with our Agile development approach.

A typical ticket in our project was an encapsulation of a user story, which consisted of a title, a one line description, value, acceptance criteria, and subtasks. In the first half of development, we also used effort-oriented metrics (story points) to measure the amount of work incurred by taking on a ticket, but, as previously mentioned, we later decided to abandon this aspect. An example of a ticket can be seen on Figure18.

mara42 commented on Nov 17 2019 • edited • + ...

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

Value

- Working product

Acceptance criteria

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

SP:?

Subtasks:

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

Figure 18: Ticket Example (source: <https://github.com/>)

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 the master branch, excluding merge commits.

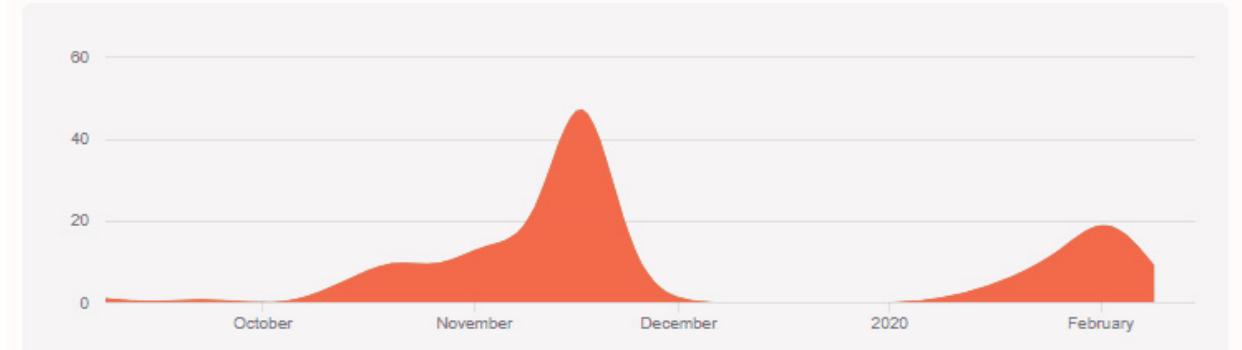


Figure 19: Contributions to the Master branch (source: <https://github.com/>)

Nevertheless, this rigour in creating tickets was not always achievable. In particular, it was difficult to create sufficiently small tickets at the beginning of the development process when more fundamental components of the systems were being developed. In these cases, we assigned tickets to a pair or group of people. This approach achieved the following:

- Improved the overall code quality and fastened production [53].
- Minimised review time over the long run.
- Distributed the knowledge of large system components across a few people instead of one person.
- Eased introducing the new team member to the project.

In addition, we used a scrum board to get an overview of ongoing tickets. Although the Scrum community is engaged in an ongoing discussion about the benefits of a physical scrum board over an online one [54], given

we had no actual workplace, using the former was not possible. The board allowed us to see which tickets needed to be reviewed and which ones 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 Figure20.

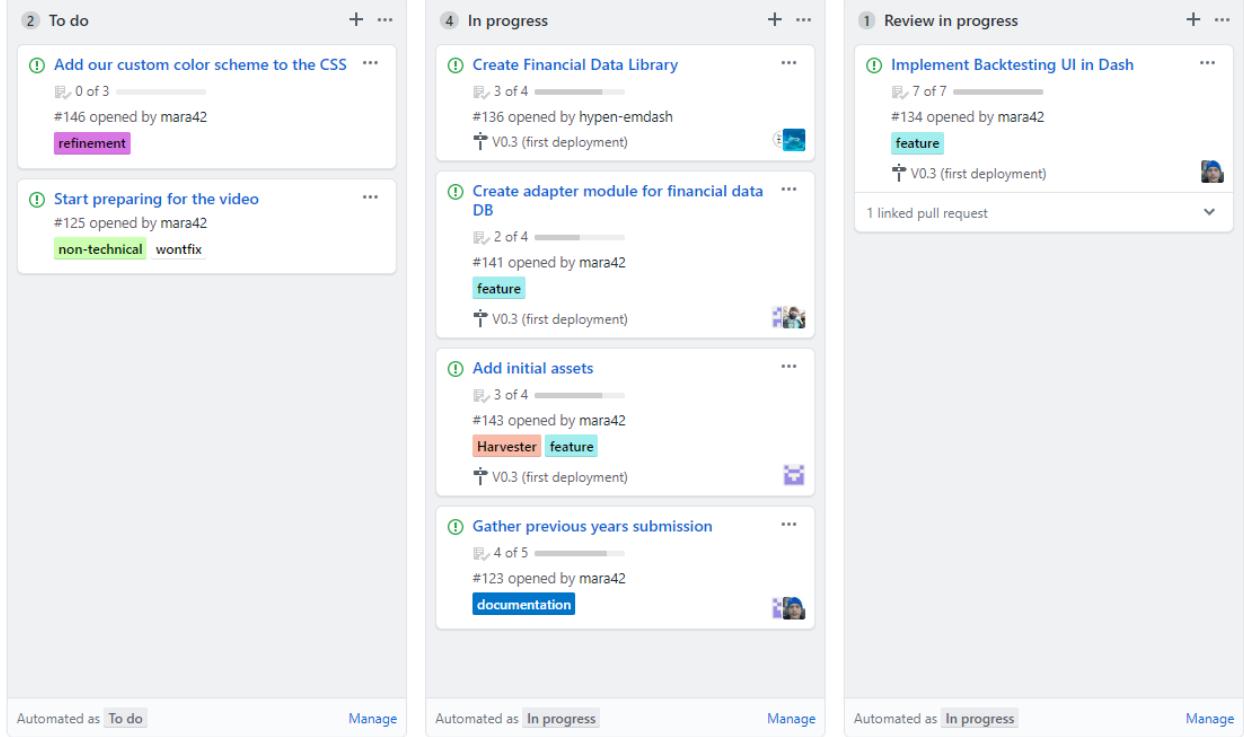


Figure 20: Scrum board - truncated

The details of the workflow largely depended on the task at hand and the person working on it. The decision whether to do pair-programming or to work individually was left up to individual members. Before we commenced coding, the team agreed on a development environment and coding standards (for a detailed analysis, see 6.4). Since this report was a relevant portion of the workload, we decided to treat it as code. The report was written in L^AT_EXwith an overall structure of the document defined in a main.tex file that linked to individual sections. This allowed members to work on different sections concurrently just as they could with the codebase.

Regardless of the nature of a ticket, the week, or in case of some larger tasks two week long cycle, ended with its creator indicating that the changes were ready to be integrated. This was done by publishing the changes and creating a new pull request, which - upon approval - pushed the changes onto the master branch of our project. To ensure code quality, we set up a continuous integration (CI) environment (more on that in 6.4). After all checks passed, the changes were successfully integrated into the codebase.

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 proved to be harder than we anticipated. The team identified the two major options available for Python to be Django and Flask. Although we decided to use Django for our first semester’s minimum viable product (MVP), we had ended up spending a significant portion of our available time learning how to use it. At the beginning of the second semester, we had to decide whether we were

would continue using Django or switch to 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 and therefore allows us to choose the layout of the code ourselves.
- Flask comes with the bare minimum features required for web-development, which means that we don't need to manage the complexity introduced by features we're not using. Django on the other hand comes prepackaged with a more complete feature-set. This would be desirable in a large web application, but introduces significant overhead in cases like ours, where our website has only a handful of pages.
- Django all but insists on using its ORM for all database interaction. This conflicted with our plan to have a more manual approach through the use of the Finda module.

Thus, we chose Flask, a Python-based micro web framework. Flask provides the functionality required for building web applications, namely managing HTTP requests, using the ‘requests’ library and rendering templates [55]. Flask leverages ‘Jinja2’ as a template engine [56]. We used the ‘render-template’ library to process and parse HTML templates for all of our website’s pages, except for the dashboard, which is described in more detail below. We used some of Flask’s extensions, such as Flask-Login, which provides user session management[57] and ‘flask-wtf’, a wrapper for the ‘WTForms’ package, to manage web forms [58].

6.3.2 GUI

Dash is great for applications that require data visualisation, modelling, or analysis [31], which is precisely what is needed for our portfolio backtesting software. It allows us to create reactive single-page apps, which means that with the use of tables, drop-down menus, and other forms of input, the app is updated instantly. This is done with the use of ‘callbacks’ [59]. Callbacks allow us to exchange data with our web server asynchronously through AJAX requests whenever a user input changes. Upon receiving a response to such a request, the corresponding output is updated immediately without refreshing the page.

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

```

1 def register_add_portfolio_button(dashapp):
2     dashapp.callback(
3         Output("portfolios-container", "children"),
4         [Input("add-portfolio-btn", "n_clicks")],
5         [State("portfolios-container", "children")],
6     )(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 is called automatically. This also happens when the Dash page is first launched. Therefore, it is common to start the called function by catering to this case and raising a ‘PreventUpdate’ exception if the input parameter is set to ‘None’. An example can be seen in [60].

The called function then receives the input properties as parameters. Furthermore, we can add any number of parameters as ‘State’ objects as demonstrated above. When returning from a function call, the same number of values specified in the callback must also be returned. Mismatching the number of return values results in a runtime error. Mismatching the input parameters, on the other hand, yields a different function signature, and so the callback is not registered with the desired function. Doing this does not raise any errors.

Fortunately, Dash offers great developer tools [61], such as the ‘Callback Graph’ visible on Figure21, which can be accessed by running the application in debug mode and activating the Dev tools. This graph proved very useful, especially for the reasoning presented in 6.3.2.

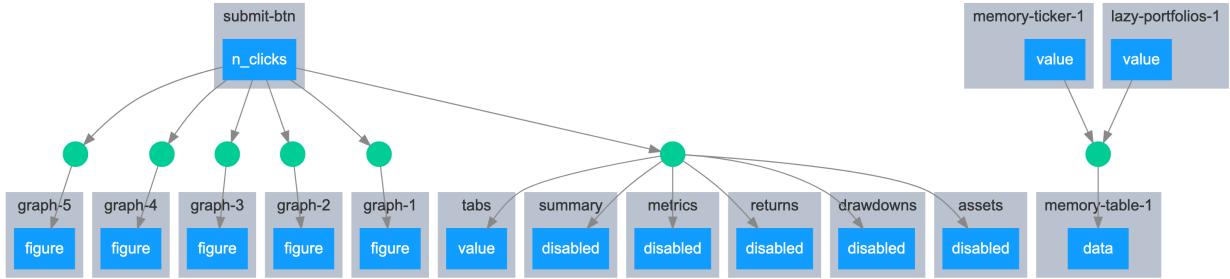


Figure 21: Callback Graph on Thalia

Limitations

Although Dash is fairly intuitive and well suited for one-page applications, it has its limitations when trying to dynamically create components. To be more precise, all callbacks need to be registered before runtime. 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 were forced to try and find a general solution to this problem. Many approaches have been suggested by the Dash Community, some even by the development team [62]. In the end, we decided to use the most stable approach we identified, generating all components beforehand and hiding them from the user until activated. This means that the relevant function now also has the responsibility of returning the visibility of the required ‘div’ element. Note that this is a highly requested feature in the Dash community, and is likely to be added soon, but for now, we are forced to resort to workarounds.

Furthermore, the data we show in the dashboard is centred around the strategy object discussed in 5.4. At the beginning of the 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 other methods. Although there exist ways to do so [63], we were not satisfied with any of the approaches listed. Consequently, we decided to go forward with the original approach, which is also the approach favoured by Dash, and combine multiple outputs into one function. Although this introduced additional complexity to our code, it had a positive overall effect on the application’s performance.

Layout

The layout of the application, as discussed in 6.3.2, 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 display elements were composed using Python. The library is then responsible for generating 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 [64]. 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’, an open-source CSS framework [30]. Bulma is ideal for creating responsive elements, as it automatically adapts the website to different screen sizes [65]. It is also compatible with most popular web browsers. Another advantage of Bulma is that we can create a very modern-looking website quickly since the framework automatically does a lot of work for us.

It also provides pre-styled elements, components (such as the navigation bar) and alerts for the users. Additionally, it is intuitive to learn when compared to ‘Bootstrap’, as Bulma uses memorable class names, such as ‘.button’, and has a very straightforward modifier system. It consists of nothing but CSS stylesheets without using any JavaScript and is therefore highly compatible with a framework like 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 6.3.2.

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

- Specification of a portfolio as a set of pandas Dataframes, the common data exchange format in our application.
- Calculation of a portfolio's return over time.
- Regular contributions to 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 [66] and bt [67]), each of these was found to be 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'. For each of its functions, Anda takes as input a 'Strategy' object that specifies the entire list of parameters needed to run 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 to our specific needs and avoid having to produce complicated wrapper code for existing backtesting libraries.

6.3.4 Database and 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 week long 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. [68]

- **Type:** Since the project constraints specify we use SQL queries, only relational DBMSs 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 [69] [70]. 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 [69].
- 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. As such, it 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 [71] [72]. 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 role of the Data Harvester is to collect live data from multiple sources, parse it to a format usable by the Anda library and store it with the Finda module. In addition to this, it contains a mechanism for managing collected data, so that our data collection can be adapted to user demands. Finally, it is responsible for making sure that all API requests made are compliant with the API's specified limitations, as breaching these could result in a ban and therefore haemorrhage data collection.

The following are the requirements our team identified for the Data Harvester module:

- **Continuous Updating Mechanism:** The database has to be updated regularly to contain live data.
- **Redundancy:** As we are interfacing with third party software, redundancy measures must be implemented so that the Harvester can continue to gather live data in the event of an API call failing.
- **Maintenance:** Financial markets are dynamic and new financial instruments are constantly being created. Therefore, it must be easy for the team to add or remove assets from those tracked by the Harvester.
- **Standard Data Format:** To maintain data integrity and successfully interface with the Finda module, all data must be parsed to a standard format without sacrificing numerical accuracy or granularity.
- **Data Interpolation:** Some financial assets require interpolation to account for days on which data is unavailable (e.g. weekends and public holidays when most exchanges are closed).
- **Security:** Since the Data Harvester interfaces with external APIs, it must not introduce any significant vulnerabilities that may jeopardize Thalia's operation.

6.3.6 Continuous Updating Mechanism and Maintenance

This module of the Data Harvester was assigned the responsibility of regularly updating the database. It consists of a group of closely coupled components each fulfilling a specific role. These are:

- **The run_updates script** Running this script triggers a round of updates. It contains data for the specific configuration necessary to poll each API (e.g. the location of the API access key to be used and the number of calls to be made).

- **The update procedure** Runs a series of methods in turn. Firstly, each supported API is checked for availability. For each asset assigned to that API, the module then fetches the appropriate amount of data. If the asset has no data available, all data is collected from a fixed start date onwards. During this stage, the module also tracks the number of calls made to enforce compliance with each API's terms and conditions. Whenever possible, the Data Harvester makes sure that data is updated as soon as possible so that the data stored is as close to being fully live as can be.
- **An initialiser** This sub-module loads and handles the persistent data specifying what assets and asset classes to query APIs for. Persistent data is stored in CSV files so as to be human-readable. This makes it relatively easy to add support for new assets to the Harvester.

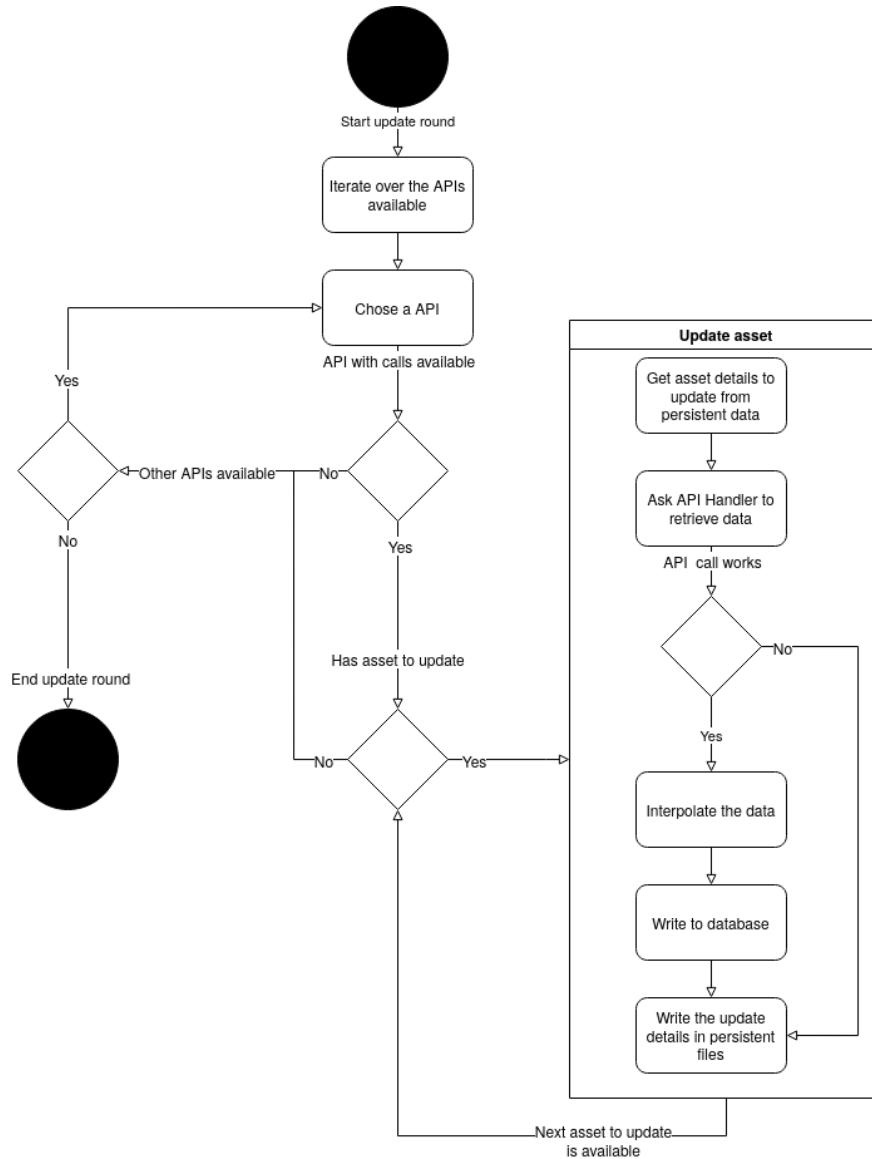


Figure 22: Flow chart for a round of updates[2]

6.3.7 Redundancy

Our current architecture allows for an asset to be served by several APIs. In the event of an API failure, the asset's data can still be collected from another available source. This makes it simple to integrate a high

degree of redundancy for popular assets into our product.

6.3.8 Standard Data Format

As previously discussed, all data written to the database is stored in pandas Dataframes. The Dataframes abide by the following format:

```
1 Columns: [AOpen<Decimal.Decimal>, AClose<Decimal.Decimal>,
2           AHHigh<Decimal.Decimal>, ALow<Decimal.Decimal>,
3           IsInterpolated<Integer>]
4
5 Index: [AssetTicker<String>, ADate <datetime.date>]
```

Listing 5: Pandas Data Frame Format

Data standardization is conducted as soon as the data has been received from the API. Each API is handled by a separate API Handler object. In addition to selecting what API to query for a specified asset, this object ensures the data gathered has been converted to the appropriate format. The API Handler also logs any potential changes made to the data's format from the provider's side.

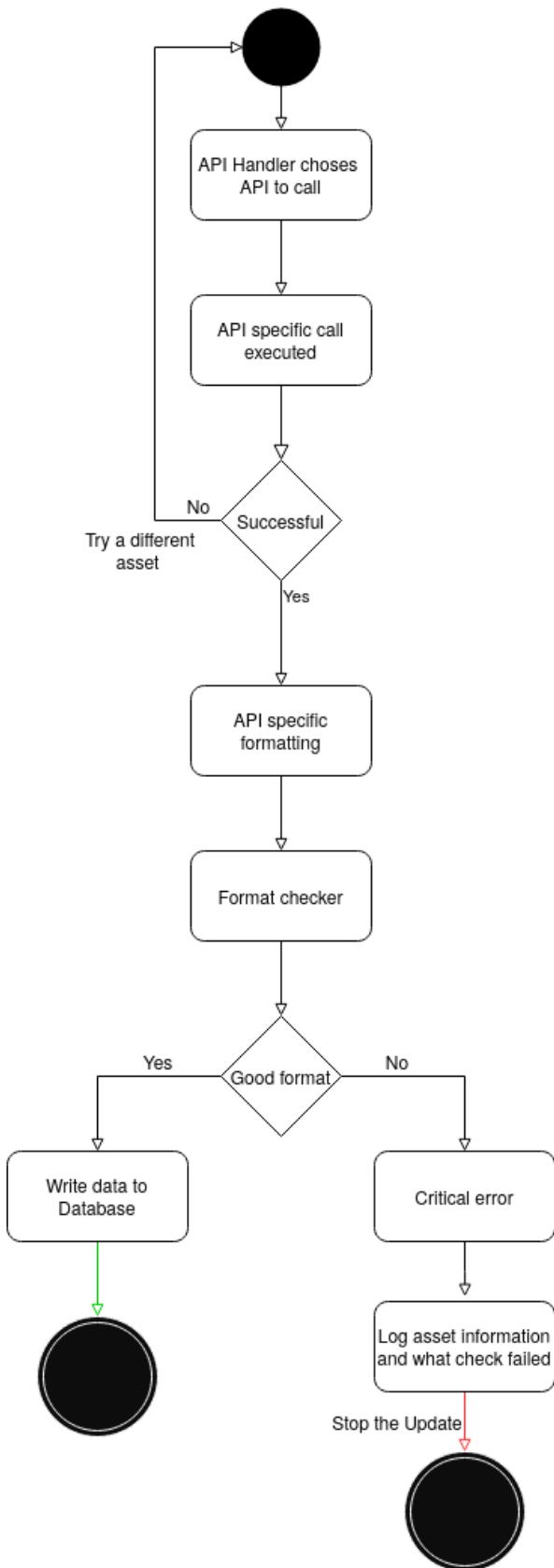


Figure 23: API Handler Flow Chart [2]

6.3.9 Data Interpolation

The numerical analysis performed by the Anda library can only be sensibly run on continuous sets of data. Real world data is often messy and needs to be cleaned before it is suitable for further processing. In the case of asset prices, the form that this ‘messiness’ manifests itself in are periods for which an asset isn’t traded and therefore does not have a price. Additionally, data contained by APIs may not be complete and may therefore contain gaps for which assets prices simply aren’t recorded. The team decided to assign the responsibility of managing this uncertainty to the Data Harvester as it is loosely coupled to the other layers of our system. The Harvester manages such gaps through the process of nearest neighbour interpolation, where the last known price is assumed to hold for the entire duration of the gap. This practice is standard when considering financial data, and is how major exchanges report prices during weekends and bank holidays. Interpolated dates are flagged in the database to make sure this process is reversible.

The following is an example of how such data is stored:

Pre-Interpolation:

Index	ADate	AHigh	ALow	AOpen	A Close	AssetTicker	IsInterpolated
3	2000-11-16	126.86	126.89	126.86	87.36	VFIAX	0
4	2000-11-17	126.44	126.44	126.44	87.07	VFIAX	0
5	2000-11-20	124.12	124.15	124.15	85.47	VFIAX	0
6	2000-11-21	124.55	124.45	124.55	85.78	VFIAX	0

Post-Interpolation:

Index	ADate	AHigh	ALow	AOpen	A Close	AssetTicker	IsInterpolated
3	2000-11-16	126.86	126.89	126.86	87.36	VFIAX	0
4	2000-11-17	126.44	126.44	126.44	87.07	VFIAX	0
5	2000-11-18	126.44	126.44	126.44	87.07	VFIAX	1
6	2000-11-19	126.44	126.44	126.44	87.07	VFIAX	1
7	2000-11-20	124.12	124.15	124.15	85.47	VFIAX	0
8	2000-11-21	124.55	124.45	124.55	85.78	VFIAX	0

6.3.10 Security

In the future, we expect to collect data from sources that are not available free of charge. Additionally, our offering of financial data is key to our unique value proposition. It is therefore easy to see how a potential data breach could end up having catastrophic financial and legal implications for our company. To mitigate this risk, we decided to conduct an internal security audit of the Data Harvester module to identify any potential vulnerabilities. To do so, we created and analysed a scenario where an attacker is attempting to disrupt the operation or gain control of the Data Harvester. Our analysis yielded the following security concerns that needed to be preemptively addressed:

Security Concerns

1. **API calls:** The data received from an API may be intercepted or modified.
2. **API keys:** An attacker could gain control of API keys if not stored safely.
3. **Rogue APIs:** An API can send a malicious package instead of the expected data.

We addressed these concerns in the following ways:

Solutions

1. **API calls:** Depending on the API settings, we will use either an IPSec policy or a VPN connection. The details of this should first be discussed with the financial data provider.
2. **API keys:** We will salt and then hash the API keys with SHA512 before storing them. Since the number of financial data APIs on the market is relatively small, we can use the SHA512 algorithm without worrying about computation time.

3. **Rogue API:** We have added several checks to ensure data retrieved is what was expected. Should the Harvester receive malformed data, a critical error will be raised and the incident will be logged for later inspection.

6.4 Integration and Deployment

Despite our focus on writing well documented, high quality code, the process of integrating our system's various components proved to be difficult. In the first term, the team spent a great deal of time on integration as many components of the system had difficulty working together. Since this term consisted of significantly more coding and less work on non-technical deliverables, we were able to significantly expand the suite of DevOps tools put in place to aid with this process [73].

We decided to redesign the development environment from scratch instead of building directly on last term's. The list of libraries required by our project can be found in the 'setup.py' file in the following lines of code:

```

1 """A setuptools based setup module."""
2 from os import path
3
4 from setuptools import find_packages, setup
5
6 here = path.abspath(path.dirname(__file__))
7
8 install_requires = [
9     "flask",
10    "flask-login >= 0.5",
11    "flask-migrate",
12    "flask-wtf",
13    "pandas",
14    "dash",
15 ]
16
17
18 tests_require = ["pytest", "coverage", ]
19
20 extras_require = {"dev": ["black", "flake8", "pre-commit"], "test": tests_require}
21
22 setup(
23     name="Thalia",
24     version="0.2.0",
25     packages=find_packages(),
26     install_requires=install_requires,
27     extras_require=extras_require,
28 )

```

Listing 6: setup.py - Development environment

One of the first decisions we had to make was to choose a standard coding style. For this, we used 'flake8', which can be seen amongst the extras in the code snippet above. The original documentation of flake8 defines it as "[...] 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" [74]. However, as is common practice, we also decided to redefine the maximum line-length as 88 instead of 79, as we found this convention to be a hindrance when writing code.

Another tool used to enforce uniform style was the 'black' auto-formatter for Python [75], which formatted the code for us upon every save (if enabled) and when committing code to GitHub. This has been achieved by the use of githooks [76], which are programs that are triggered upon certain git actions. Setting up these hooks required the 'pre-commit' package. This ensured that both flake8 and black were guaranteed to run before publishing changes.

6.4.1 Continuous Integration

Many studies have investigated the positive effects of developing in a continuous integration (CI) environment (see, for example, [77] and [78]). 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 had also passed its tests. Note that, in this section, we will only discuss testing as a part of CI and not the testing strategy (for this, see 7).

Another important part of the DevOps toolchain is the use of containers, which is what Docker helps to achieve [79]. 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 [80]. When choosing the CI environment, Docker support was one of the main requirements.

The most promising candidate for this was CircleCI [81], 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 Docker container or virtual machine for the code.
2. Checks previously cached requirements (for more detail see Figure24).
3. Installs the requirements from ‘requirements.txt’
4. Caches the requirements for faster future 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 AWS, see 6.4.2

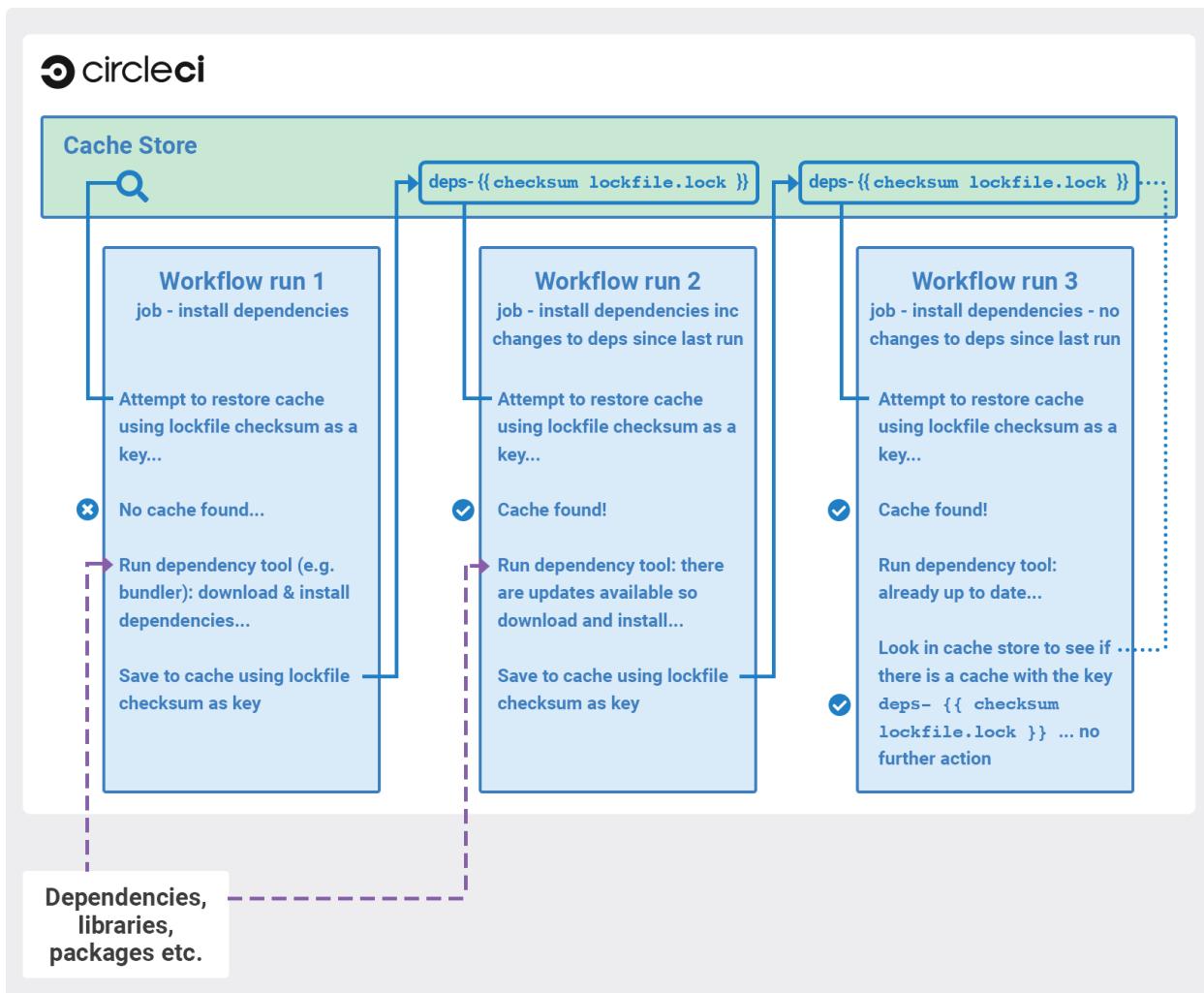


Figure 24: CircleCI on Caching (source: <https://circleci.com/docs/2.0/caching/>)

The outcome of these steps is visible on the CircleCI web platform, but - more importantly - also on GitHub. CircleCI will prevent a merge with master if failing tests (or no tests) have been found.

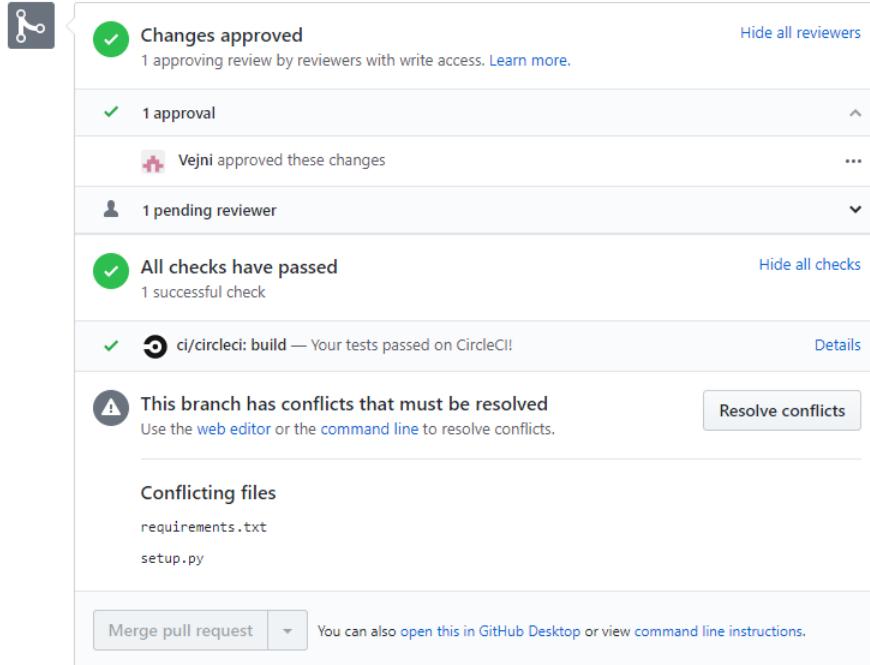


Figure 25: CircleCI on GitHub (source: <https://github.com/>)

With these steps, 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 of the literature concerning deployment reveals a plethora of different strategies for deploying and hosting web applications [82]. Our decision of how to choose among them involved the following considerations:

- Price - since our budget is severely constrained, 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 solutions involving deployment to a virtual machine in the Cloud.

Due to native CircleCI support and a free-tier service, we initially chose Heroku [83] as our hosting provider. This allowed us to host our application for free in the nascent stages of development while providing ample opportunity for horizontal and vertical scaling later on if required.

However, it quickly became apparent that Heroku was not appropriate for our purposes, due to its lack of support for sqlite databases [84]. Hence, we moved to AWS [85], which is also supported by CircleCI [86].

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” [87]. Using AWS 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 6.4.1 to ensure the validity of the current codebase state.

3. If this step is successful, the master branch is pushed to a remote repository recognized by AWS via git.
4. AWS executes the ‘Procfile’ script stored in the root of our project to start the application using a ‘gunicorn’ web server [88].

Our deployment process is thus fully automated 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 Figure26.

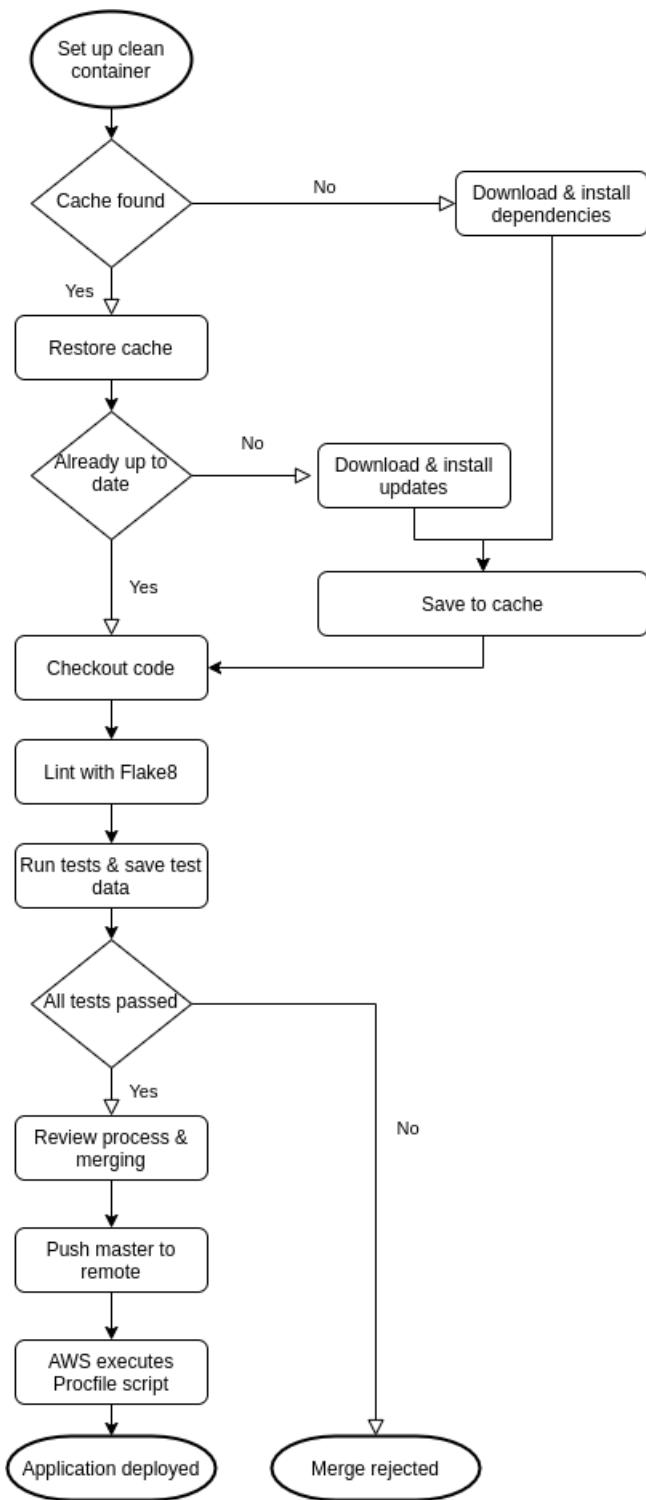


Figure 26: CI/CD workflow

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 [89]. 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 [90]).

Our testing strategy so far consisted of three main types of tests: 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 APIs 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 which is faster.

```
Tests/
|-- Finda
|   |-- conftest.py
|   |-- finData.db
|   |-- helpers.py
|   |-- test_fd_manager.py
|   |-- test_fd_read.py
|   `-- test_fd_remove.py
|       `-- test_fd_write.py
`-- Harvester
    `-- Thalia
        |-- conftest.py
        |-- test_callbacks.py
        |-- test_factory.py
        |-- test_user.py
        `-- test_views.py
`-- analyse_data
    |-- test_anda.py
    `-- test_data
        |-- AAPL.csv
        |-- AAPL_dividends.csv
        |-- BRK-A.csv
        |-- MSFT.csv
        |-- MSFT_dividends.csv
        `-- risk_free_rate.csv
```

Figure 27: 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 our 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 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. Moreover, 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 were 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 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 prototype 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.

```

1 INFO:root:
2
3 Started update at: 2020-03-09 19:49:13.691662
4 INFO:root:Started updating yfinance at 2020-03-09 19:49:13.691850
5 INFO:root:Updating for yfinance doing 1 calls
6 INFO:root:calling for:
7 asset_class: index_funds ticker: VFIAX start_date : 1970-1-1 end_date: 2020-03-08
8 INFO:root:api returned <class 'pandas.core.frame.DataFrame'>
9 INFO:root:Received data frame with data between 2000-11-13 - 2020-03-08
10 INFO:root:Start interpolationdataframe shape: (4857, 7)
11 INFO:root:Data Frame Interpolateddataframe shape: (7054, 7)
12 INFO:root:Writing interpolated dataframe to DB
13 INFO:root:Finished updating yfinance at 2020-03-09 19:49:33.868327
14 INFO:root:Started updating nomics at 2020-03-09 19:49:33.868475
15 INFO:root:Updating for nomics doing 1 calls
16 INFO:root:calling for:
17 asset_class: currency ticker: ALL start_date : 1970-1-1 end_date: 2020-03-08
18 INFO:root:api returned <class 'pandas.core.frame.DataFrame'>
19 INFO:root:Received data frame with data between 2018-05-31 - 2020-03-08
20 INFO:root:Start interpolationdataframe shape: (648, 7)
21 INFO:root:Data Frame Interpolateddataframe shape: (648, 7)
22 INFO:root:Writing interpolated dataframe to DB
23 INFO:root:Finished updating nomics at 2020-03-09 19:49:36.457801
24 INFO:root:Finished all updates at: 2020-03-09 19:49:36.457934

```

Figure 28: 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

```

/usr/lib/python3.8/site-packages/werkzeug/local.py:378: in <lambda>
    __getitem__ = lambda x, i: x._get_current_object()[i]
-----
self = <SecureCookieSession {'_fresh': True, '_id': '3da5fe10fd627b080e6c2d3525f339fe2ca24ae875ef9b44ed1e2c3aefd26cf5daf4
eb44a705e2fa7723ae6a7cec477948bad9cbd569357bfb219b430b3b733', 'user_id': '1'}>
key = '_user_id'

def __getitem__(self, key):
    self.accessed = True
>   return super(SecureCookieSession, self).__getitem__(key)
E   KeyError: '_user_id'

/usr/lib/python3.8/site-packages/flask/sessions.py:84: KeyError
===== 2 failed, 46 passed in 30.72s =====

```

Figure 29: Example of pytest assertion introspection

2. **Mock:** The Mock [91] 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 [92] 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 text 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 [93] 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, i.e. the prices of assets, tends to experience frequent and unpredictable fluctuations. Part of our testing strategy was the generation of 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 monotonically 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 by 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 [94]. 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 [95].

8 Evaluation

8.1 Approach

Evaluation of our product began after the team finished development and testing. As mentioned previously, 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 [96]. 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 [97]. 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.

8.2 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 Figure30.

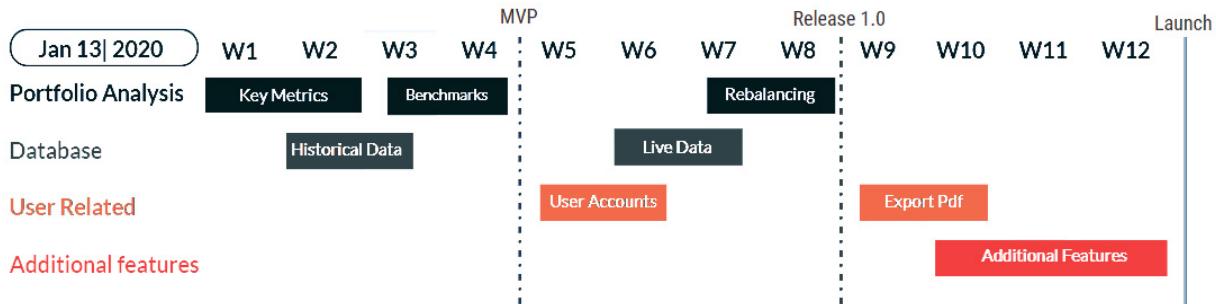


Figure 30: Technical Report: Roadmap - revisited

Due to our team's focus on correctness 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 two were properly implemented.

The functional requirements not implemented at the time of writing are the ability for a user to search for assets by category when creating a portfolio and front end support for currency conversion. Their omission is a result of a lack of resources, architectural limitations and our internal assessment of their importance. Halfway through development, the team assessed each functional requirement not yet implemented and determined these two to be of low importance. This decision was based on the fact that searching assets by category 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 to the Thalia Web and Finda modules, meaning that its implementation

would represent a bad cost to value ratio when compared with other remaining features. Although back-end support for different currencies is implemented in the Anda module, front-end support would require sourcing and collecting a large amount of new data. The team decided that front-end support for this feature is not critical for a prototype and would prefer implementing it later on when adding support for languages and other elements of localization. Even without these features, we believe this prototype to be fully featured and usable and therefore ready for the next stages of development.

8.3 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.

8.3.1 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 financial 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 basic instructions on how to use Thalia is featured in the navigation bar on Thalia's landing page. Additional material is also available in the user manual 11.
- All major functions must be visible from the initial landing page.
All pages are accessible through the navigation bar 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.

8.3.2 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%[98].
- 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.

8.3.3 Performance

Performance and load times of Thalia's web page hosted on our cloud hosting provider were tested with the use of Google Lighthouse [99], 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.

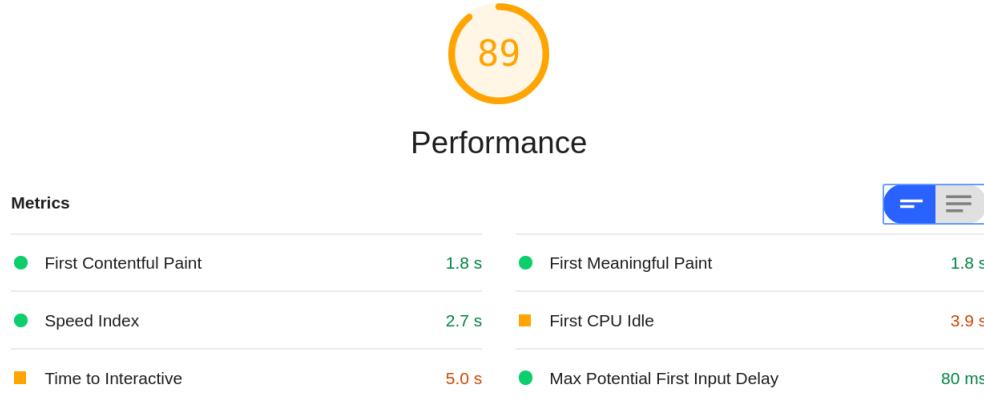


Figure 31: Results of Lighthouse Performance Tests

- 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. That said, 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.

8.3.4 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 12 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 11.*
- 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, hence 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 which is merely a high quality industrial prototype.*

8.3.5 Implementation

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

8.3.6 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 [100].
- The Data Gathering Module must not exceed its contractual usage limits.
Fulfilled as previously stated.

8.3.7 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.

8.4 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 development dependencies can be installed additionally, but are not necessary for running Thalia.

8.4.1 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 [101]. In addition, our disclaimer is based on disclaimers found in the terms of service of competing software [102] 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 (unknown) 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.

```

python -m piplicenses --summary
Count License
1 Apache
2 Apache 2.0
6 BSD
1 BSD License
6 BSD-3-Clause
1 BSD-like
1 Dual License
1 LGPL
17 MIT
1 MPL-2.0
1 UNKNOWN

```

Figure 32: 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[103].
- Provided services should not constitute financial advice under UK law to avoid being subject to financial advice legislation and potential liability.
Thalia is merely a tool for running simulations, and as such we at no point advise users on how to invest their real-life assets. This is also stated explicitly in our legal disclaimer.

8.4.2 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 different 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 [104]. 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.

8.5 Evaluation Results

The prototype of Thalia we have delivered along with this report fulfills the vast majority of identified functional requirements. We have also provided adequate reasoning for why we decided not to implement the remaining functional requirements. Where 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.

9 Conclusion and Further Work

9.1 Conclusion

In this report, we have presented Thalia, a portfolio backtesting web application. After engaging in discussion of the product requirements and our approach to development, we have provided justifications for any design and implementation decisions as well as technology choices. In the following sections, we then went on to specify our testing strategy and the results of our evaluation process.

As a consequence of the preceding evaluation section, we perceive our project to have been successful. We have delivered a high quality, functioning piece of software that fits our identified business needs and market niche. Since we have already provided extensive discussion of whether our requirements have been met, we will now provide an outline of further work to be done moving forward.

9.2 Further Work

9.2.1 Payment Service Integration

Charging for the service we provide requires the processing of customer payments. Given the sensitive nature of payment data (e.g. credit card numbers), our preferred approach is to outsource processing of payments to a third-party service, such as Stripe [105]. Our responsibility would then be reduced to that of integrating the vendor payment system into our codebase.

9.2.2 Implementation of a Scripting Language

One of our optional features from last semester’s technical report was the development of a scripting language that would allow investors to specify dynamic trading strategies. As an example, an investor may want to sell their equities for bonds whenever there is a drawdown greater than 10% in equity price indices. While support for such a scripting language was outside of the scope of this project, it is technically feasible, as has been demonstrated by other backtesting tools, e.g. Stockbacktest [16].

9.2.3 Additional Risk Metrics

Our aim for this project was to provide values for the most commonly encountered risk metrics when backtesting a portfolio. However, many other, more exotic risk metrics exist. Over time, we may want to incorporate additional metrics into our dashboard to enable an even more nuanced analysis of an investment strategy. To prevent more novice users from being overwhelmed, these could be deactivated by default and enabled on demand.

9.2.4 Expanding Asset Selection

Our team is proud of the breadth of assets available for analysis provided by our tool. As discussed previously, the inclusion of European market assets sets us apart from most other commercially available tools. Nevertheless, we were unable to support every asset class and even every asset within the asset classes we support. In particular, we would be interested in supporting standard exchange traded derivatives, such as Futures and Options. In order to appeal to other investor demographics, it may also be worthwhile integrating price data for African, Asian and Latin American assets.

9.2.5 Equity Dividends

Investors receiving dividends for the equities they own face a choice: Should dividends be reinvested in the equities themselves or spread over the whole portfolio? Supporting calculations that involve dividends would enable us to provide empiric answers to these questions. Our business logic library already provides support for dividend calculations, hence this would be a simple matter of making appropriate changes to the UI.

10 Appendices

11 Appendix A - User Manual

11.1 Launching Thalia

Since Thalia is a web application, it can be accessed from your web browser at the address: <http://thaliabacktest.xyz/>.

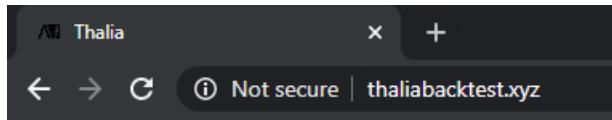


Figure 33: Thalia Web

11.2 Browsing Thalia

The application can be navigated using the navigation bar, visible on Figure34.



Figure 34: Thalia Navigation Bar

In case Thalia is launched on a different device, such as mobile, or in a smaller window on your desktop, the layout will change to fit the screen. In this case, the navigation bar becomes a so-called “hamburger button“ and dropdown as can be seen on Figure35.

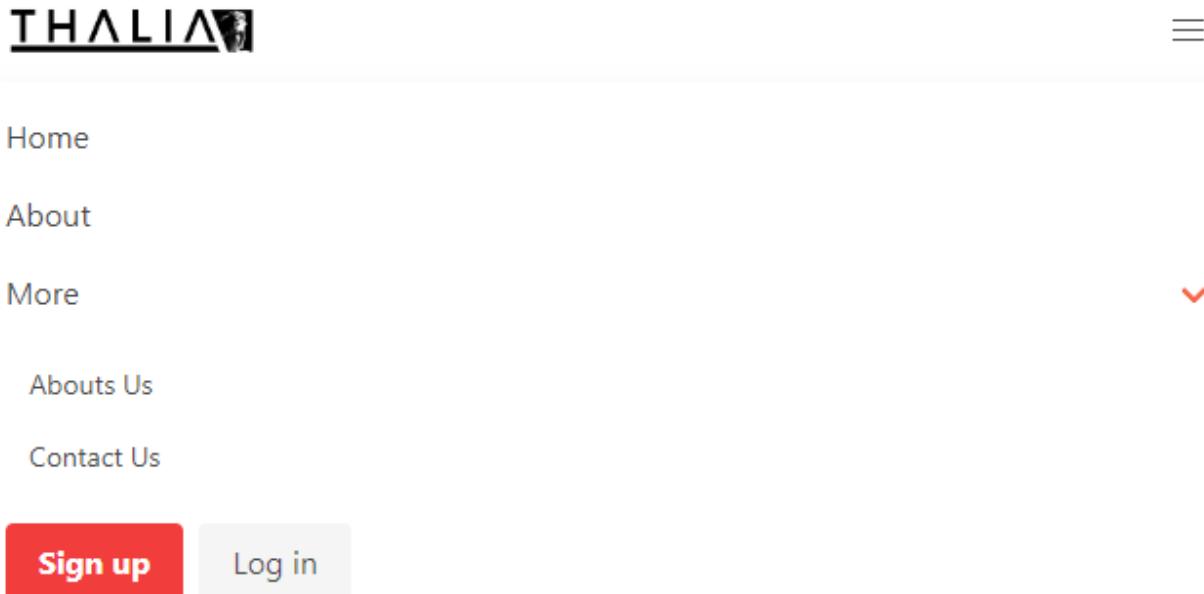


Figure 35: Thalia Navigation Bar - Hamburger

For the rest of this manual, we shall assume that the application is being accessed from a desktop computer, although the layout is much the same and equally intuitive on both desktop and mobile displays.

11.2.1 Homepage

By default, the user is directed to the homepage, although some of the other pages can also be accessed by providing the correct URL. The following image depicts the homepage:

Figure36.

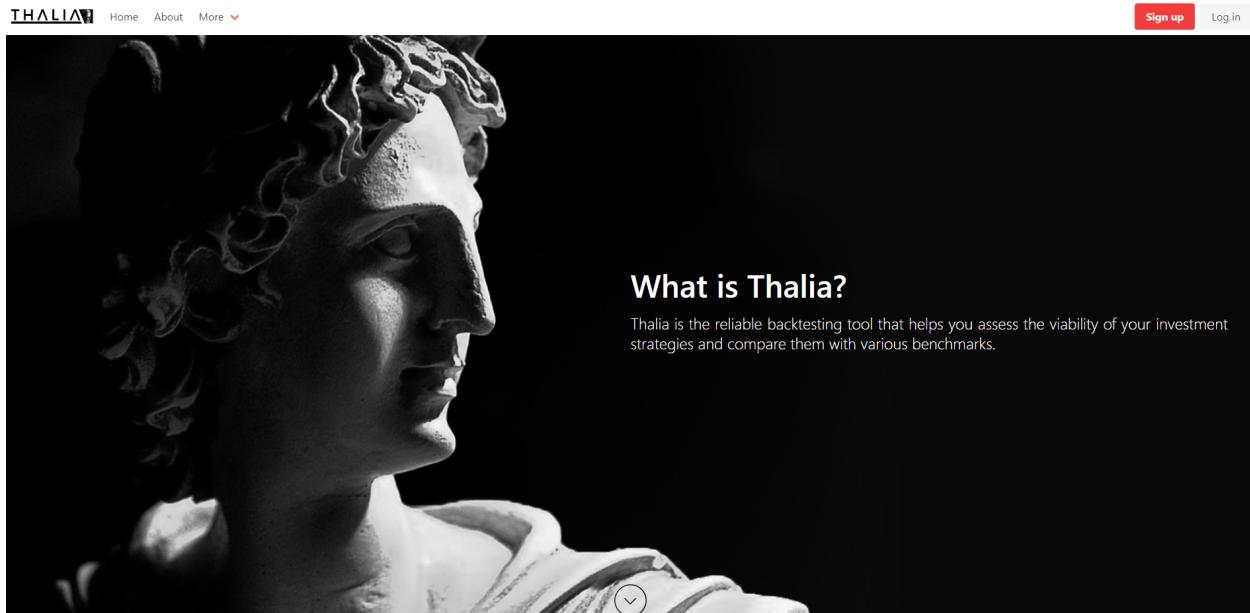


Figure 36: Thalia Homepage (source: <http://thaliabacktest.xyz/>)

The purpose of the homepage is to provide our application with a cover. When users arrive at the homepage, they are greeted by a short description of what our software can do.

As we do not wish to have users jumping blindly into portfolio analysis, we provide them with some basic information on the backtesting process. A link at the end of the description leads to the 'About' page, where users can read up on the topic in greater detail.

Scrolling further down, the user may encounter a short register form, or if the user is logged in, an embedding of our Twitter timeline. At the bottom of most pages you can find a footer with links to our social media and a short legal disclaimer to better define the service we are offering.

Register

Still with us?

Go ahead and create an account, or log in if you already have one. Once registered, you can backtest to your heart's content.

Username

Password

Confirm Password

Register

[Already have an account?](#)

[Login](#)

Already signed in?

Why not jump right in and create your dream portfolio? If you would like to stick around, make sure to follow us on our social media pages to see our latest.

Tweets by @Thalia99627941

Load more Tweets

Thalia
@Thalia99627941

Help pass the time during lockdown by perfecting your investment strategy with the **#Thalia** backtesting tool!

April 5, 2020

Embed

[View on Twitter](#)

Figure 37: Top: User not logged in; Bottom: User recognised

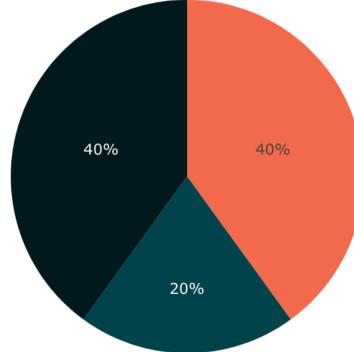
11.2.2 About Page

The about page of our application provides a more detailed description of the problem domain and is available at <http://thaliabacktest.xyz/about/>.

The user can access this page by either directly typing the URL into the address bar, by clicking on the learn more button on the homepage, or by navigating to it using the navigation bar.

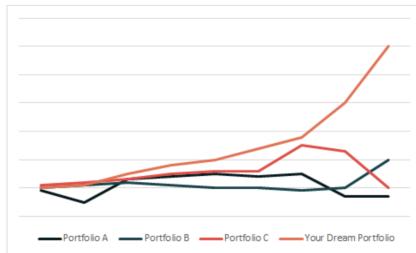
The About page looks as follows:

- Index Funds
- Stocks
- Currency



But how does it work?

Imagine you're nearing retirement and can't afford to sit patiently through another financial crisis waiting for your investments to regain the value that they have lost. You're also passionate about finance, and devised an investment strategy, which couldn't possibly fail, you say. Imagine this dream portfolio consists of the following:



That's great, but wouldn't it be useful to consider similar portfolios, comparing how much they've lost in the worst year out of the last 50? Just to make sure that the dream portfolio indeed works. This process is known as backtesting and is usually done with the assistance of a computer.

That is where Thalia can help you, with Thalia, you can:

- Specify an asset allocation using hundreds of assets across major asset classes
- Plot the performance of strategies against each other and an index benchmark
- Gain insight into a strategy's key risk and performance metrics
- Model regular contributions and re-balancing strategies
- Save and share a portfolio to compare and discuss with peers



Figure 38: Thalia About Page (source: <http://thaliabacktest.xyz/about/>)

11.2.3 Log In and Sign Up Pages

If the user is not yet logged in, links to the 'Log In' and 'Sign Up' pages are visible on the right hand side of the navigation bar as seen on Figure34 or Figure35. In addition, they are directly available at <http://thaliabacktest.xyz/login/> and <http://thaliabacktest.xyz/register/>.

Both of these forms follow a standard layout, with the login requiring:

- Username
- Password
- Remember me (optional)

For the sign up form, the fields are:

- Username
- Password (minimum eight characters, at least one letter, one number and one special character)
- Confirm Password

By convention, 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.

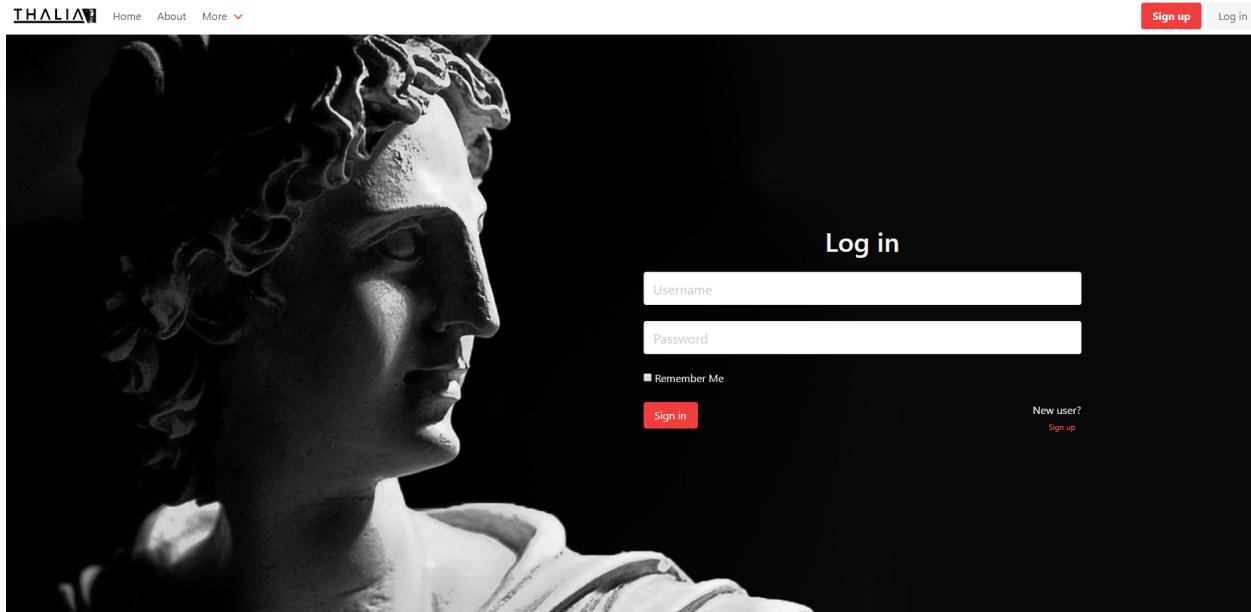


Figure 39: Thalia Log In Page (source: <http://thaliabacktest.xyz/login/>)

If the user is already logged in and attempts to access these pages, they will be redirected to the homepage. In addition, the navigation bar changes, allowing one to log out as visible on Figure40.

After clicking the log out button, the user will find themselves back on the homepage.

11.2.4 Contact Us Page

The 'Contact Us' page is available at <http://thaliabacktest.xyz/contact/> or by opening the dropdown menu on the navigation bar and clicking the link. This short form allows users to provide feedback, report issues or request new features.

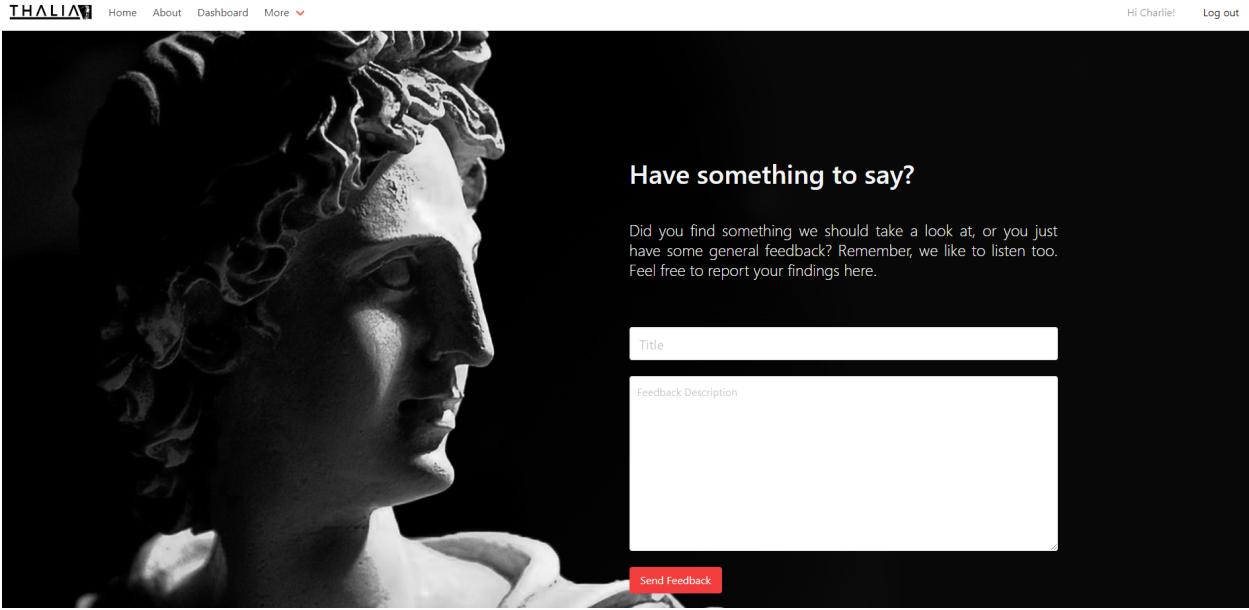


Figure 40: Thalia Report Issues Page (source: <http://thaliabacktest.xyz/contact/>)

11.2.5 About Us Page

We take great pride in our project and thus wanted to sign our work following the advice in [49]. This simple page is dedicated to the team behind Thalia.

11.2.6 My Portfolios Page

The ‘My Portfolios’ Page is only accessible to authenticated users. Here, the user may view their saved portfolios and their asset allocations. If you are running Thalia for the first time, this page will be empty, encouraging you to go create and save a portfolio. To create a portfolio, simply head to the dashboard and allocate your assets following the instructions in 11.3.2. Before submitting your investment strategy, make sure to save it for later. You can achieve this by clicking on the default portfolio name and changing it to something more fitting. Then, click the ‘Save Portfolio’ button. When you next visit the ‘My Portfolios’ Page, you can see the portfolio and its associated asset weights. Clicking on the box will take you back to the dashboard. The corresponding allocations will already be loaded to the table. Alternatively, you can select any of your previously saved portfolios directly from the dashboard by using the dropdown menu and selecting your chosen portfolio.

The following image shows the aforementioned page with some example portfolios saved.

Dream Portfolio A		Dream Portfolio B		Dream Portfolio C	
JEP	20.00%	JPY	10.00%	ALL	50.00%
KRW	40.00%	SB	70.00%	NGN	50.00%
ALL	20.00%	GTQ	10.00%		
JPY	20.00%	O	10.00%		

Figure 41: My Portfolios - Example (source: <http://thaliabacktest.xyz/gallery/>)

11.3 Running a Backtest

To run a backtest, the user must be authenticated. After a successful registration and/or login process, the user can head to the Dashboard and begin backtesting.

11.3.1 Dashboard

The Dashboard, which is at the heart of our application, is available for registered users at <http://thaliabacktest.xyz/dashboard> or via the link on the navigation bar.

This web page is divided into the following tabs:

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

At first, only the Ticker Selector Page is accessible to the user.

This is because the remaining tabs are used to display the results of backtesting one or multiple portfolios and consequently do not initially contain any data.



Figure 42: Thalia Dashboard - Disabled Tabs

Let us now consider each tab individually.

11.3.2 Ticker Selector

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

- Start Date: The day the investment is made.
- End Date: The last day of the investment, which defaults to the present date.
- Initial Amount: Initial balance. The default currency is dollars, as is convention for financial applications.

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: Defaults to Portfolio 1, Portfolio 2, etc.
- Contribution Amount: The amount that should be added to the total investments on a regular basis (e.g. after receiving a monthly paycheck).
- Contribution Frequency: Specify how regularly these contributions should occur.

- Rebalancing Frequency: Specify how often rebalancing (reestablishing the initial weights of the portfolio by buying and selling assets) should happen.
- Asset Allocations: 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 input to be typed in order to find an asset faster. The selected item is then added into the table below.

Ticker:

	AssetTicker	Name	Allocation
x	SPY		1
x	VTSAX		2
x	JPY		1

Figure 43: Thalia Dashboard - Asset Table (source: <http://thaliabacktest.xyz/dashboard/>)

The user then specifies the weight of the asset in their portfolio. This must 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 33% of the investment is allocated to asset A and 66% to asset B.

If 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 option available is to submit.

If the user has failed to enter some required data, such as an initial amount, or hasn’t selected at least one asset, a warning message is shown, telling them to complete the form correctly. Note that this also happens when the user selects a time frame shorter than a year, as many of our metrics do not make sense for investments over such short periods of time.

In many cases, the user may want to compare their portfolio to a benchmark, i.e. to a small set of standard portfolios. These can be selected from the “Lazy“ dropdown menu at each portfolio, which then populates the table with the desired proportions.

Alternatively, the user may want to test an allocation on price data they possess but that is not included in Thalia’s database. If the user has the data in the form of a CSV file, this can be accomplished by dragging and dropping the file to the corresponding field.

Upload your own ticker:

Drag and Drop or [Select Files](#)

Figure 44: User Uploaded Data Field

11.4 Viewing the Results

The results of the backtesting process are divided between the remaining tabs of the dashboard. Let us now discuss what each of them tells the user

11.4.1 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. All graphs are fully interactive. The user may zoom in on selected areas, hover over desired data points, save the plot as an image, etc. The graphs can be reset by double clicking on them.



Figure 45: Dash Functionalities

The plot seen at the top of Figure 46 shows the total return of each portfolio. Users will typically use this graph for comparison. In addition, for each portfolio the user has specified, 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 relative to the last.

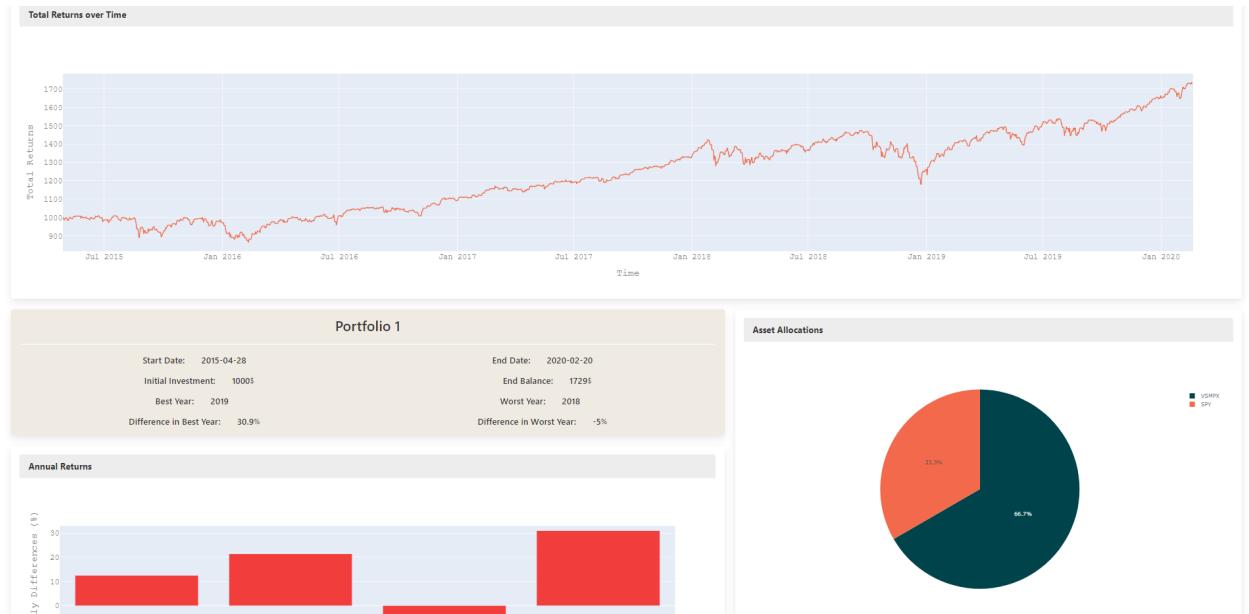


Figure 46: Summary Tab (source: <http://thaliabacktest.xyz/dashboard/>)

It is also important to note that in some cases, there is no data for a given asset. For example, there is no price data for cryptocurrencies before the 2000s as they did not yet exist. In edge cases where the portfolio is made up entirely of such assets, the strategy may not be valid over a given investment period.

If there is only one portfolio given as input, and it is affected by the above, the simulation will be run over the largest possible time frame over which data for all assets is available. In the case where the user provides multiple portfolios with different time frames, we provide output where price data is available.

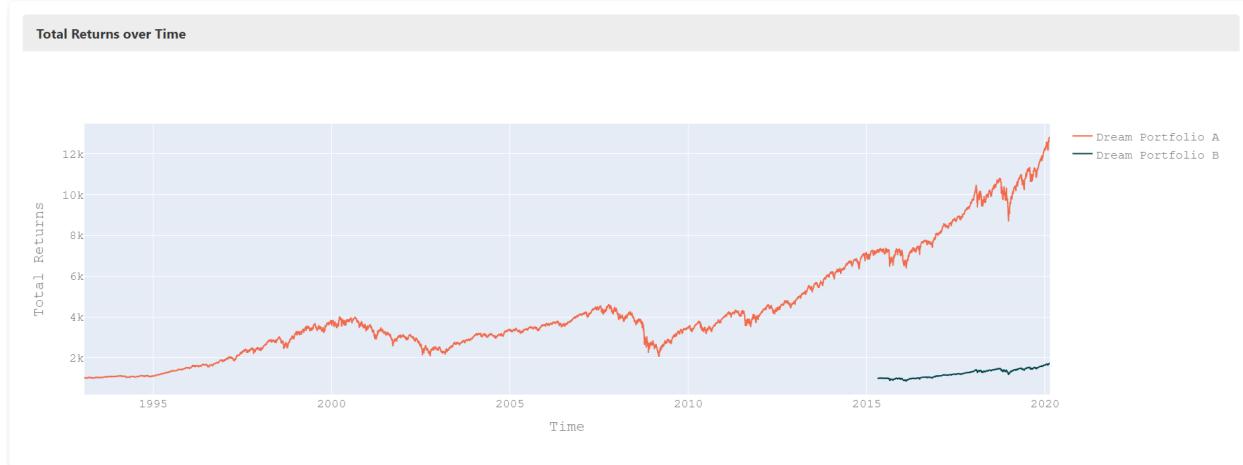


Figure 47: Different Time Frames per Portfolio (source: <http://thaliabacktest.xyz/dashboard/>)

11.4.2 Metrics

The ‘Metrics’ tab shows a simple table of key metrics for each portfolio. This allows for a quick comparison of strategies. Although this page does not show much more data than the previous one, it can easily be expanded later. In case the table is bigger than the browser window, the user can also easily scroll down to see the rest of the output.

Key Metrics

Metric	Dream Portfolio A	Dream Portfolio B
Initial Balance	1000	1000
End Balance	12737.24	1718.19
Best Year	38.05	30.79
Worst Year	-36.8	-5.16
Max Drawdown	55.19	20.16
Sortino Ratio	0.94	1.34
Sharpe Ratio	0.61	0.9

Figure 48: Metrics Tab (source: <http://thaliabacktest.xyz/dashboard/>)

11.4.3 Returns

The ‘Returns’ tab allows the user to compare the annual returns of each portfolio. It consists of a bar chart similar to the one in the Summary tab and a table similar to the one in the Metrics tab. The graph can be seen on Figure49.

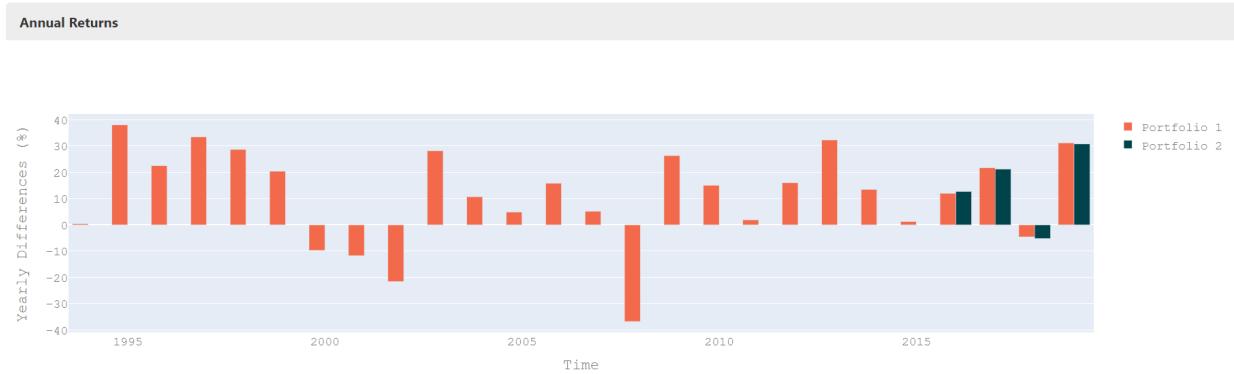


Figure 49: Metrics Tab (source: <http://thaliabacktest.xyz/dashboard/>)

It may happen that one of the portfolios is not available at certain times as there is no data for its assets. In this case, no values are shown.

11.4.4 Drawdowns

The ‘Drawdowns’ tab shows declines during the investment period. This is shown on a graph comparing each portfolio visible on Figure50.

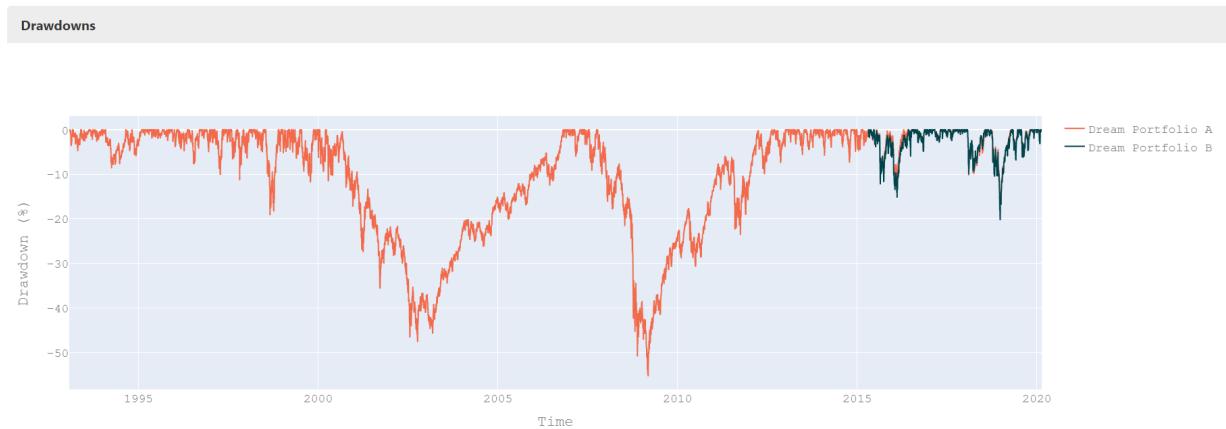


Figure 50: Drawdowns Tab - Graph (source: <http://thaliabacktest.xyz/dashboard/>)

For deeper insight, a table of the longest drawdowns is generated for each portfolio. By default, this will be the top 10 or as many as exist in case the portfolio does not have 10 drawdown periods.

Dream Portfolio A Top Drawdowns						
Drawdown (%)	Start	End	Recovery	Length	Recovery Time	Underwater Period
-55.19	10/10/2007	09/03/2009	16/08/2012	1 year, 4 months	3 years	4 years
-47.52	27/03/2000	09/10/2002	26/10/2006	2 years	4 years	6 years
-19.35	21/09/2018	24/12/2018	12/04/2019	3 months	3 months	6 months
-19.03	21/07/1998	31/08/1998	23/11/1998	a month	2 months	4 months
-13.02	21/07/2015	11/02/2016	18/04/2016	6 months	2 months	8 months
-11.7	19/07/1999	15/10/1999	17/11/1999	2 months	a month	3 months
-11.2	08/10/1997	27/10/1997	05/12/1997	19 days	a month	a month
-10.1	29/01/2018	08/02/2018	06/08/2018	10 days	5 months	6 months
-10.04	19/02/1997	11/04/1997	05/05/1997	a month	24 days	2 months
-9.3	20/01/2000	25/02/2000	17/03/2000	a month	21 days	a month

Dream Portfolio B Top Drawdowns						
Drawdown (%)	Start	End	Recovery	Length	Recovery Time	Underwater Period
-20.16	21/09/2018	24/12/2018	23/04/2019	3 months	3 months	7 months
-15.11	24/06/2015	11/02/2016	08/06/2016	7 months	3 months	11 months
-9.9	29/01/2018	08/02/2018	25/07/2018	10 days	5 months	5 months
-6.77	06/05/2019	03/06/2019	20/06/2019	28 days	17 days	a month
-6.18	29/07/2019	14/08/2019	28/10/2019	16 days	2 months	2 months
-5.9	09/06/2016	27/06/2016	08/07/2016	18 days	11 days	29 days
-4.8	08/09/2016	04/11/2016	15/11/2016	a month	11 days	2 months
-3.11	21/01/2020	31/01/2020	05/02/2020	10 days	5 days	15 days
-2.77	02/03/2017	13/04/2017	27/04/2017	a month	14 days	a month
-2.43	26/07/2017	18/08/2017	11/09/2017	23 days	24 days	a month

Figure 51: Drawdowns Tab - Tables (source: <http://thaliabacktest.xyz/dashboard/>)

11.4.5 Assets

Here, the user may encounter a pie chart similar to the one on the summary tab. However, the pie chart shows the proportions of the asset classes as opposed to the specific assets allocated. This is crucial information for the user as it can show how diverse their portfolio truly is.

Assets Breakdown

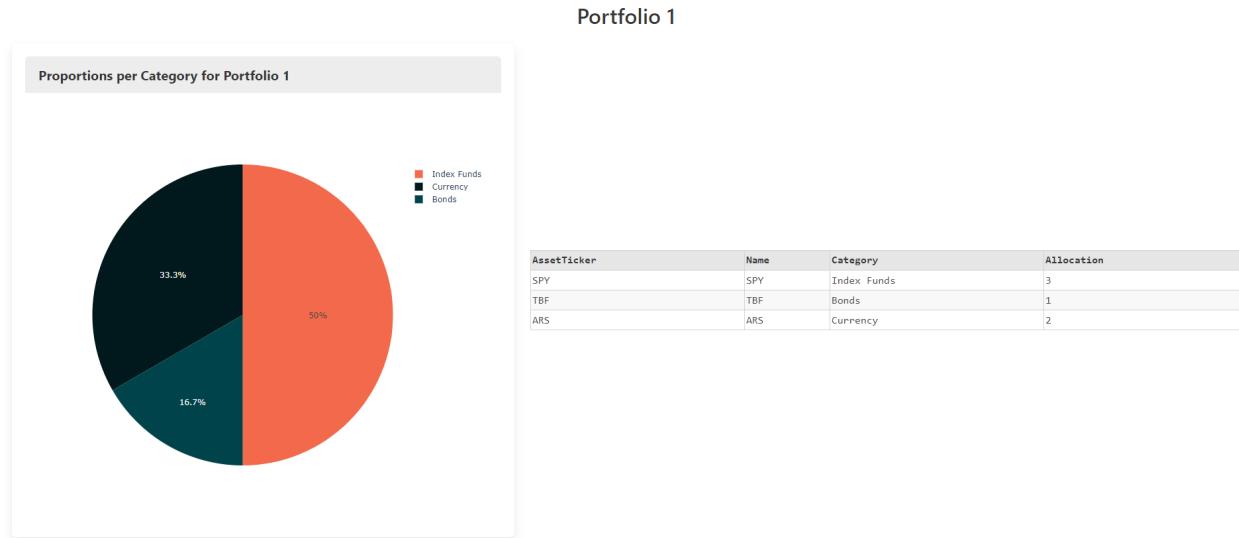


Figure 52: Assets Tab (source: <http://thaliabacktest.xyz/dashboard/>)

11.4.6 Overfitting

Our last results tab encompasses a relatively complex feature, testing for overfitting. Here, the user may test if any of their portfolios exhibits bias to particular market conditions, for example by choosing a specific timeframe over which one or more assets performs significantly better than usual. For example, a simulation showing the prices of Microsoft stock in the 90s would hardly indicate the viability of such a portfolio

nowdays. Overfitting is dangerous as it may skew the investor's perception of the risk-reward characteristics of a portfolio. By clicking the button, the user may opt to run an overfitting test. If it fails to detect a tendency towards overfitting in any of the selected portfolios, the following message is displayed:

The screenshot shows a horizontal navigation bar with six tabs: Ticker Selection, Summary, Metrics, Returns, Drawdowns, Assets, and Overfitting. The Overfitting tab is highlighted with a red background. Below the navigation bar, the word "Overfitting" is displayed in bold black font. A large orange button labeled "Check Overfitting" is centered. At the bottom, a blue box contains the text "We have not detected overfitting in any of your portfolios."

Figure 53: Overfitting Tab (source: <http://thaliabacktest.xyz/dashboard/>)

Alternatively, if overfitting is detected, it will warn the user, highlighting the corresponding portfolio(s).

12 Appendix B - Maintenance Manual

12.1 Prerequisites

Before running Thalia on a local setup, the following prerequisites must be fulfilled:

- The project must be run on a supported operating system, i.e. one of: MacOS, Ubuntu, AWS Linux 2, or Debian. While it also seems to run on Windows 10, there are no strict correctness guarantees as the test suite has been developed exclusively for UNIX flavoured operating systems.
- Python ≥ 3.8 must be available.
- The Python virtual environment module ‘venv’ and the ‘pip’ package installer must be installed.
- One of either Chrome or Firefox must be available for accessing the website.
- To run Selenium tests, you will need both Chrome and Firefox installed, with both ‘geckodriver’ and ‘chromedriver’ available in order to run tests in headless mode. You can find more information about this in the Selenium directory’s ‘README.md’.

12.2 Installation

In order to install Thalia, execute the following instructions in order of listing. The instruction assume a Unix shell. If you’re running Windows, please substitute instructions as needed.

1. cd into the project directory
2. Create a virtual environment (<https://docs.python.org/3/tutorial/venv.html>) and activate it:

```
python3 -m venv venv && source ./venv/bin/activate
```

3. Install the Python dependencies via pip (<https://docs.python.org/3/installing/index.html>):
`pip install -r requirements.txt`

12.2.1 Registering a Database

There are two ways to create a database of financial assets: use a pre-made database .db file or create your own database by using the Harvester.

Using an existing Database

Place the database into the ‘Finda’ directory. Rename it to `asset.db`. You may now register it with Finda by running the `register_findb.py` script.

Creating a Database with the Data Harvester

Create your own database with the Data Harvester will take some time due to the large number of API calls involved. You will need an API key for the Nomics API. Once you have acquired the necessary key, place it in a file called ‘nomics’ within the ‘DataHarvester/apis_access’ directory. Then, open a new terminal and follow these instructions:

1. Run `python first_run_dh.py` to create, register and seed the financial data database.
2. Start the harvester with `python run_harvester.py`, which will run forever.
3. To see what assets have been updated, have a look at the ‘DataHarvester/persistent_data’ directory.
4. You can start and stop the Harvester at any time. When restarted, it will continue the updating session as long as any updates remain.

Once all the assets have been updated at least once, you can proceed to the next section. This may take up to one day since we must abide by contractual API limits.

12.3 Running the application in a Development Environment

To run the project in a development environment from a modern unix shell, you can use the `start.sh` script. After running it, you will be able to access the website from `localhost:5000`.

12.3.1 Running the application in a Production Environment

Procedures to run the application in production differ from the ones listed for a development environment. Follow these instructions to setup Thalia on a web server.

- Install the ‘gunicorn’ HTTP server [88].
- Run the following command from a Unix terminal
`sudo gunicorn --bind 0.0.0.0:80 --daemon main:server`

The website should now be accessible via the standard HTTP port 80 from your IP address.

12.4 Test instructions

The following instructions have to be executed to test the application:

1. If you have not done so already, install the project requirements by using
`pip install -r requirements.txt`
2. You will also need to install additional development and testing dependencies by using
`pip install ".[test,dev]"`
3. Python and Selenium tests must be run separately. The former can be achieved by issuing
`python -m pytest --ignore=../Tests/Selenium`
4. To run Selenium tests, run the application in a development environment to ensure it is available at `localhost:5000`. You may then test it by using
`python Tests/Selenium/main_test.py`

12.5 Source Code Files

The following sections explain the responsibilities of every source code file.

12.6 Anda

- `analyse_data/analyse_data.py` – The library that handles all backtesting calculations.

Tests

- `Tests/analyse_data/test_anda.py` – Tests for `analyse_data.py`
- `Tests/analyse_data/test_data` – Test data used by `test_anda.py`

12.7 Data Harvester

- `DataHarvester/apis_access/` – Contains all our API keys for the Harvester’s raw data retrieval.
- `DataHarvester/dhav_core/api_class.py` – Abstracts individual API service methods for data retrieval from other parts of code.
- `DataHarvester/dhav_core/data_harvester.py` – Main module for the package.
- `DataHarvester/dhav_core/init_finda_db_structure.py` – Seeds database with current assets and meta-data.

- DataHarvester/dhav_core/init_update_list.py – Script for resetting the accumulated persistent data.
- DataHarvester/dhav_core/initialization.py – Initial setup required for other modules in package.
- DataHarvester/dhav_core/logger_class.py – Logging for Harvester.
- DataHarvester/dhav_core/run_updates.py – Script used to setup API keys and batch sizes for a Harvester update cycle.
- DataHarvester/dhav_core/tickers/ – Directory contains the source files for assets in different assets classes. The files dictate our supported assets in production.
- run_harvester.py – Script for running harvester on a timed interval.
- first_run_dh.py – Data Harvester Initializer, run once.

Tests

- Tests/Harvester/test_api_caller.py – Tests for API handling.
- Tests/Harvester/test_interpolation.py – Tests for financial data interpolation.
- Tests/Harvester/interpolation_result.csv – Expected interpolation results.
- Tests/Harvester/to_interpolate.csv – Data to be interpolated during tests.

12.8 Finda

- Finda/asset.db – If only using the basic setup scripts, asset.db is the assumed main financial database.
- Finda/dbSchema.sql – SQL schema for the financial database.
- Finda/fd_manager.py – Database controller for the financial data database.
- Finda/fd_read.py – Methods for reading from financial data database.
- Finda/fd_remove.py – Methods for deleting from financial data database.
- Finda/fd_write.py – Methods for writing to financial database.
- register_findb.py – Script used for ensuring the financial database has been registered, run once.

Tests

- Tests/Finda/conftest.py – Setup and tear down code for Finda tests.
- Tests/Finda/finData.db – Test database.
- Tests/Finda/helpers.py – Testing helper methods for Finda.
- Tests/Finda/test_fd_manager.py – Tests for fd_manager.
- Tests/Finda/test_fd_read.py – Tests for fd_read.
- Tests/Finda/test_fd_remove.py – Tests for fd_remove.
- Tests/Finda/test_fd_write.py – Tests for fd_write.

12.9 Thalia

12.9.1 Dashboard

- Thalia/dashboard/callbacks/allocations.py – Controller for portfolio creation, saving and retrieval.
- Thalia/dashboard/callbacks/assets.py – Controller for the asset contribution tab.
- Thalia/dashboard/callbacks/callbacks.py – Collects all other callbacks for easy importing.
- Thalia/dashboard/callbacks/dashboard.py – Controller for running a backtest and all side effects of running a backtest.
- Thalia/dashboard/callbacks/drawdowns.py – Controller for the drawdowns tab.
- Thalia/dashboard/callbacks/metrics.py – Controller for the metrics tab.
- Thalia/dashboard/callbacks/overfitting.py – Controller for overfitting check.
- Thalia/dashboard/callbacks/returns.py – Controller for returns tab.
- Thalia/dashboard/callbacks/summary.py – Controller for the summary tab.
- Thalia/dashboard/tab_elements/ – Directory contains UI generation code.
- Thalia/dashboard/tab_elements/elements.py – Generic elements shared by multiple tabs.
- Thalia/dashboard/tab_elements/lazy_portfolios – A list of formatted pre-made portfolios.
- Thalia/dashboard/user-data – Directory contains the temporary user uploaded CSVs.
- Thalia/dashboard/config.py – Commonly used constants used in the dashboard codebase.
- Thalia/dashboard/layout.py – Declaration of the main UI elements.
- Thalia/dashboard/portfolio_manager.py – Methods for storing and retrieving portfolios from the database.
- Thalia/dashboard/strategy.py – Adapter for passing user input to Anda.
- Thalia/dashboard/tabs.py – Tab element configuration and declarations.
- Thalia/dashboard/user_csv.py – Back-end code for handling temporary user uploaded CSVs.
- Thalia/dashboard/util.py – Methods for accessing Finda in the dashboard.

Tests

- Tests/Thalia/callbacks/test_allocations.py – Tests for allocation tab.
- Tests/Thalia/callbacks/test_dashboard.py – Tests for elements across dashboard.
- Tests/Thalia/callbacks/test_summary.py – Tests for summary tab.
- Tests/Thalia/test_user_csv.py – Tests for user uploaded CSV handling.
- Tests/Thalia/test_portfolio.py – Tests for portfolio storage.

12.9.2 Flask

- Thalia/models/portfolio.py – The Portfolio ORM module. Handles user saved portfolio interactions with the database.
- Thalia/models/user.py – The User account ORM module. Handles user account interactions with the database.
- Thalia/static/ – Contains static assets, images, scripts, and stylesheets used in the Flask parts of the website.
- Thalia/templates/ – Contains Jinja2 templates for different parts of the Flask website. Includes the base, homepage, authentication page, info pages, and user portfolio gallery templates.
- Thalia/__init__.py – Module responsible for the app factory of the Flask app. Passes complete Flask apps to web workers such as gunicorn. Includes Flask app setup, configuration, dash app registrations and protection, database registrations, extension setup.
- Thalia/extensions.py – Module for initialising different Flask extensions.
- Thalia/findb_conn.py – Module for accessing Finda. Separated from other modules to avoid circular imports.
- Thalia/forms.py – Declarations of different HTML form controllers.
- Thalia/views.py – Main UI logic for Flask code. Handles routing and controlling user interactions with the Flask code.
- app.dbexport – The user data database.
- config.py – Flask app configuration declarations. Includes development defaults in case user has not defined the necessary environmental variables.
- feedback.csv – User submitted feedback from the website.
- main.py – Used as the ‘main’ method for the project. It provides the results of the app factory to be passed to a server.

Tests

- Tests/Thalia/conftest.py – Setup and teardown code for Flask tests.
- Tests/Thalia/test_factory.py – Test app factory pattern.
- Tests/Thalia/test_user.py – Test user account handling.
- Tests/Thalia/test_views.py – Integration tests for views.py.

12.9.3 Selenium Tests

- Tests/Selenium/contact_test.py – Test contact form.
- Tests/Selenium/dashboard_test.py – Test dashboard.
- Tests/Selenium/login_test.py – Test login.
- Tests/Selenium/main_test.py – Convenience script to run all Selenium tests.
- Tests/Selenium/navbar_test.py – Test navigation bar.
- Tests/Selenium/README.md – Instructions and info on Selenium tests.

- Tests/Selenium/register_test.py – Registration tests.
- Tests/Selenium/social_test.py – Tests for social media links.
- Tests/Selenium/util.py – Utilities used by other Selenium tests.
- Tests/Selenium/webdriver_test.py – Test for checking whether Selenium configuration is working

12.10 Miscellaneous

- .circleci/config.yml – Defines our continuous integration jobs for CircleCI.
- .flake8 – Style guide rules for linting Python code.
- pyproject.toml – Style rules for automatic code formatter.
- pytest.ini – Default rules for pytest to run our tests.
- README.md – Short instructions on installation, tests and other notes.
- requirements.txt – Our project's python requirements with versions locked and tested for compatibility.
- setup.py – Python setup file with more loose requirements. Includes requirements necessary for testing and contributing.
- start.sh – Commonly used script for running project in a development environment.
- .pre-commit-config.yaml – Pre-commit hook.

13 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 are designed to perform well in most market conditions and thus 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 a 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 of a portfolio 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.
[<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.
[\[https://www.investopedia.com/terms/f/fixedincome.asp\]](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.
[\[https://www.investopedia.com/terms/c/commodity.asp\]](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.
[\[https://www.investopedia.com/terms/f/forex.asp\]](https://www.investopedia.com/terms/f/forex.asp)
- Overfitting - Overfitting is a modelling error that occurs when a function is too closely fit to a limited set of data points. In the context of investing, a portfolio that performs significantly better on a limited time-frame compared to a larger one is said to be overfitted on that timeframe.
[\[https://www.investopedia.com/terms/o/overfitting.asp\]](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.
[\[https://www.investopedia.com/terms/l/leverage.asp\]](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 analysing statistical trends gathered from trading activity, such as price movement and volume.
[\[https://www.investopedia.com/terms/t/technicalanalysis.asp\]](https://www.investopedia.com/terms/t/technicalanalysis.asp)
- Drawdown - A drawdown is a prolonged decline during a specific period for an investment, trading account, or fund. A drawdown is usually quoted as the percentage between its peak and subsequent trough.
[\[https://www.investopedia.com/terms/d/drawdown.asp\]](https://www.investopedia.com/terms/d/drawdown.asp)

14 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 data for both historical and live prices as well as data on dividend payouts for equities.
- Finda - An interface to our database. Provides functionality for registering, creating and connecting to databases as well as reading and writing data to them. The name has been chosen as the abbreviation of 'financial data'.
- Thalia Web - Shorthand for the Thalia Web Server providing the interface to our service for customers to interact with.

15 Appendix E - UML Diagrams

Of the UML Diagrams presented below, only those related to the Anda library show the parameter types of methods as well as the return types. This is because we believe that discussion of Anda requires more detail, since it deals with nuanced differences between numerical types such as ‘Decimal’ and ‘float’.

15.1 Anda

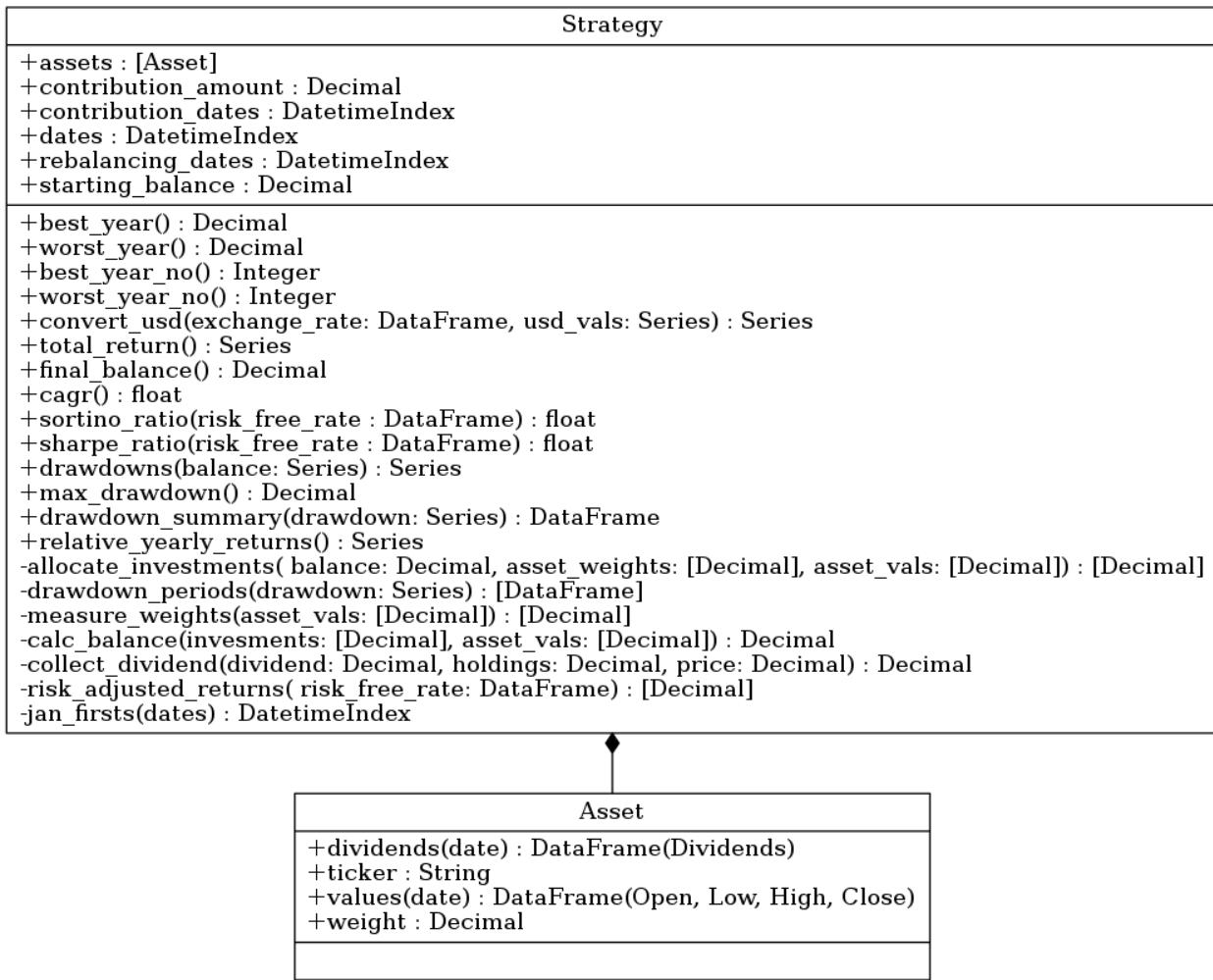


Figure 54: UML Diagrams - Anda

15.2 Data Harvester

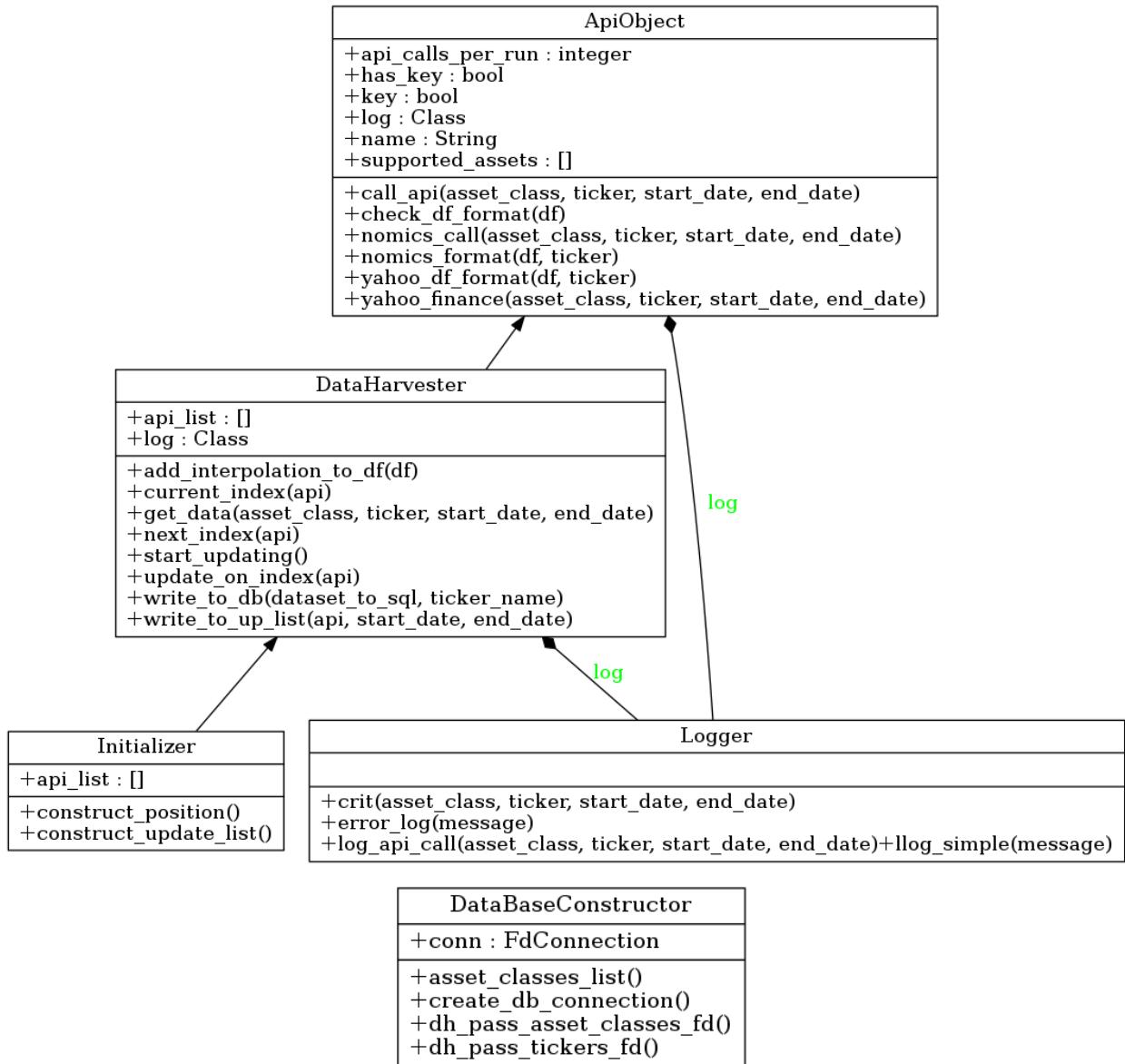


Figure 55: UML Diagrams - Data Harvester

15.3 Finda

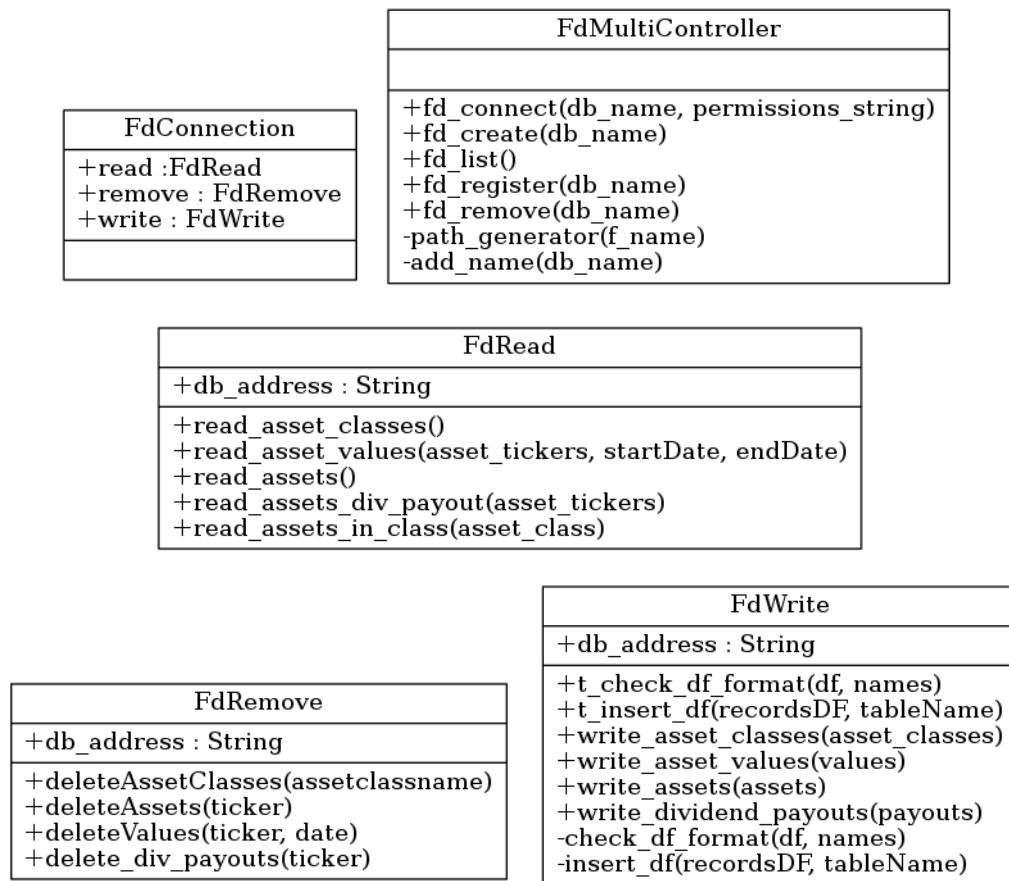


Figure 56: UML Diagrams - Finda

15.4 Thalia Web

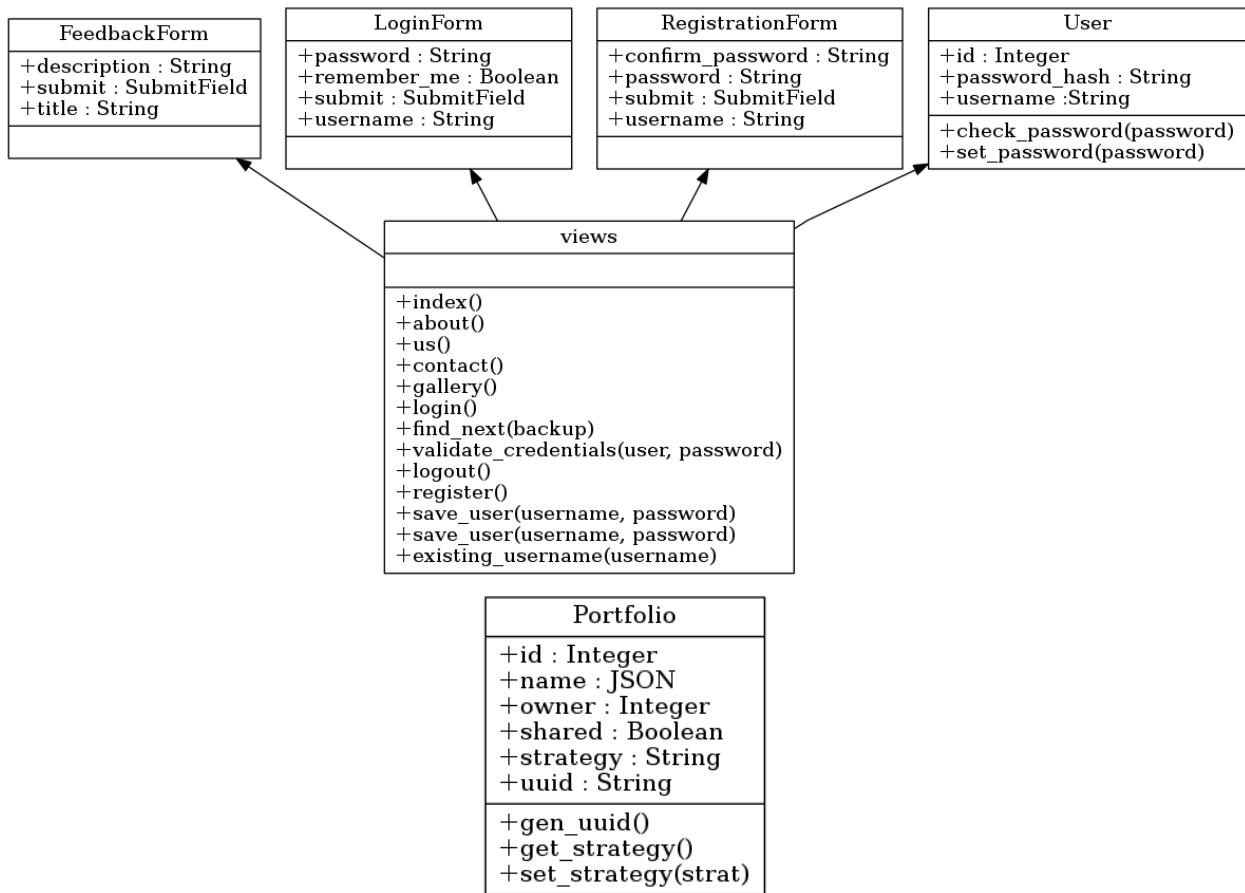


Figure 57: UML Diagrams - Thalia Web

16 Appendix F - User Types Diagrams

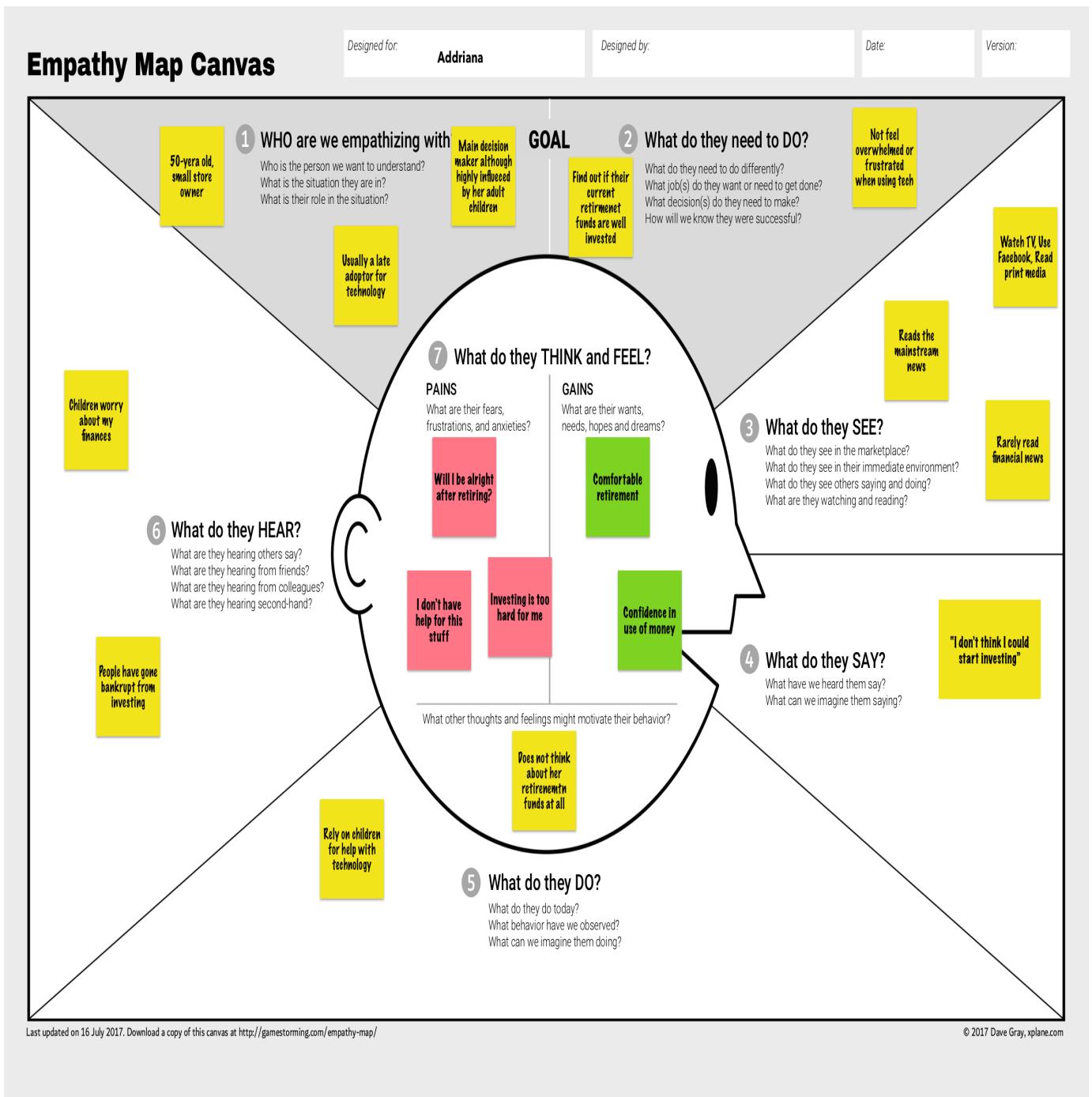


Figure 58: Addriana Empathy Map

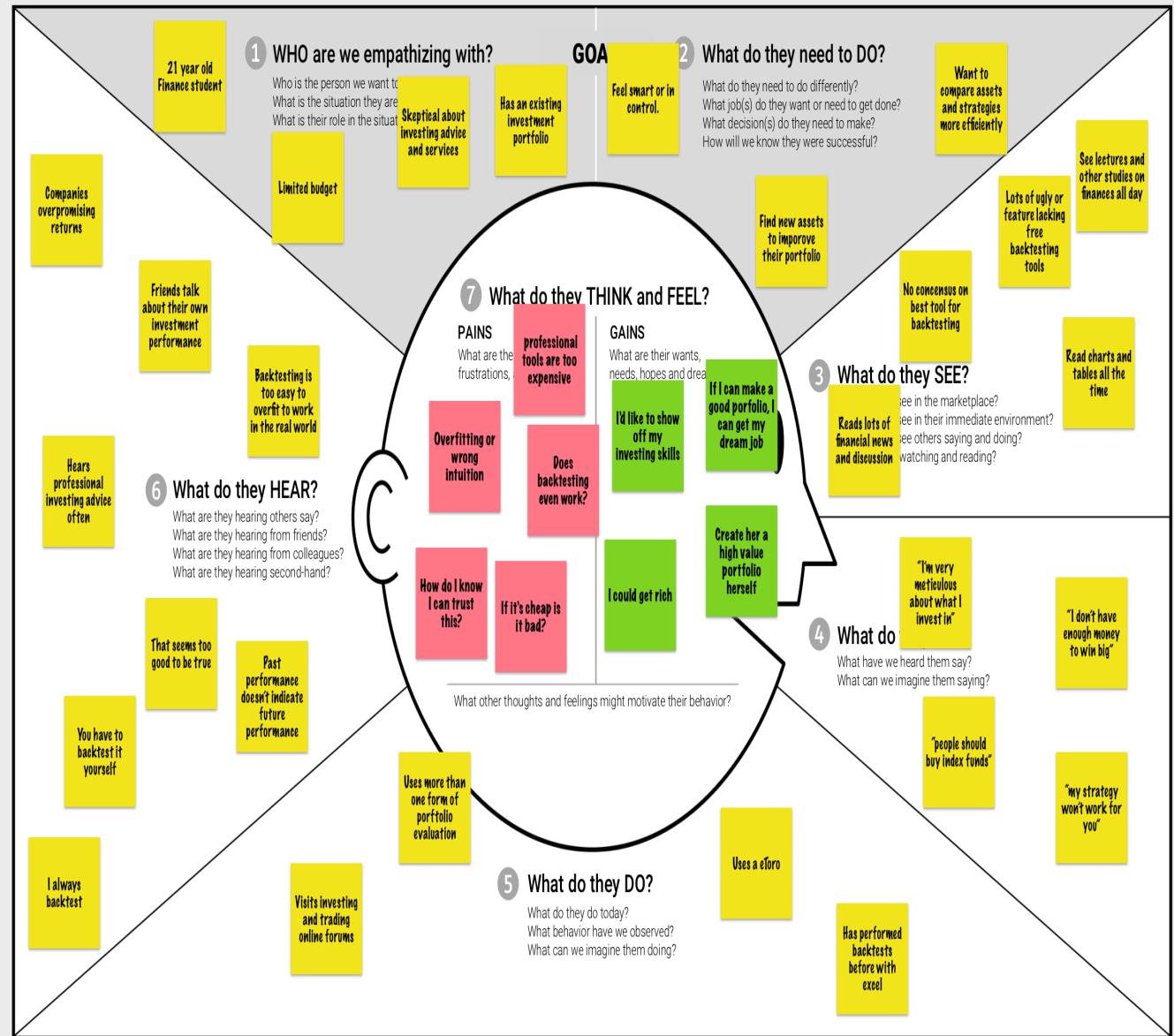
Empathy Map Canvas

Designed for: Allison

Designed by:

Date:

Version:



Last updated on 16 July 2017. Download a copy of this canvas at <http://gamestorming.com/empathy-map/>

© 2017 Dave Gray, xplane.com

Figure 59: Allison Empathy Map

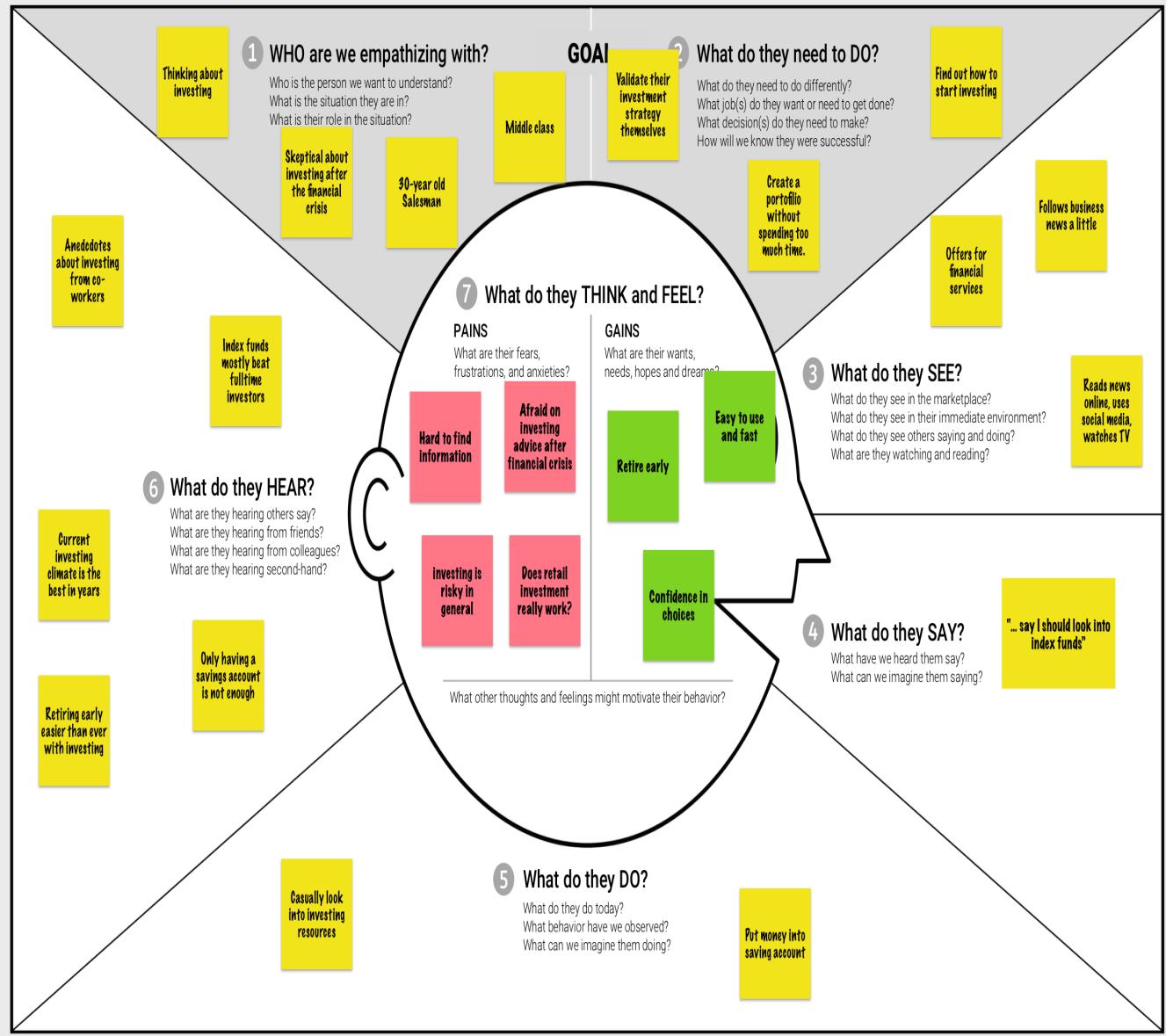
Empathy Map Canvas

Designed for: Sam

Designed by:

Date:

Version:



Last updated on 16 July 2017. Download a copy of this canvas at <http://gagelab.com/empathy-map/>

© 2017 Dave Gray, xplane.com

Figure 60: Sam Empathy Map

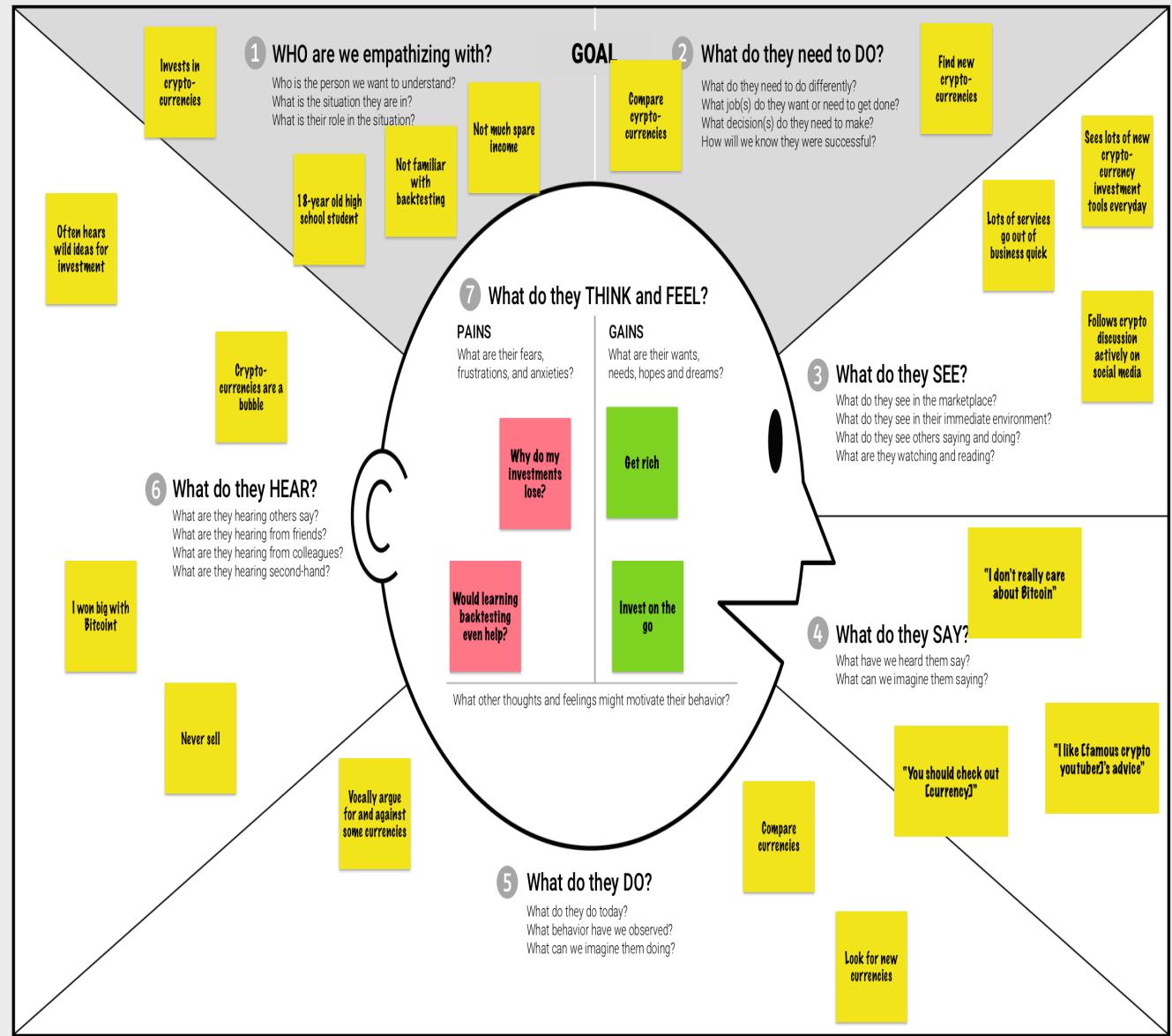
Empathy Map Canvas

Designed for: Timmy

Designed by:

Date:

Version:



Last updated on 16 July 2017. Download a copy of this canvas at <http://gagelab.com/empathy-map/>

© 2017 Dave Gray, xplane.com

Figure 61: Timmy Empathy Map

PERSONA

Personas are used to capture the type of people involved in and using your service. They are not necessarily a representation of an individual, but show the characters, opinions and qualities of a group of users. Try to create 5 personas associated with your service.

What do they look like? Draw/create an image here.



Photo by Artem Beliaikin on Unsplash

WHAT ARE THEY MOST LIKELY TO SAY ABOUT THE SERVICE?

“

I don't understand

”

Try out investing

WHAT WOULD THEY USE THIS SERVICE FOR?

NAME:

Addriana

AGE:

50

OCCUPATION:

Small store manager

LIKES:

Simple things

DISLIKES:

Technology

FAMILY AND FRIENDS:

Single



Figure 62: Addriana Persona

PERSONA

Personas are used to capture the type of people involved in and using your service. They are not necessarily a representation of an individual, but show the characters, opinions and qualities of a group of users. Try to create 5 personas associated with your service.

What do they look like? Draw/create an image here.



Photo by Max Ilienerwise on Unsplash

WHAT ARE THEY MOST LIKELY TO SAY ABOUT THE SERVICE?

“

easy and cheap option

”,

NAME:

Allison

AGE:

21

OCCUPATION:

Finance Student

LIKES:

Finance, mobile UI

DISLIKES:

Lack of feedback

FAMILY AND FRIENDS:

Finance students

WHAT WOULD THEY USE THIS SERVICE FOR?

Help validate strategy



Figure 63: Allison Persona

PERSONA

Personas are used to capture the type of people involved in and using your service. They are not necessarily a representation of an individual, but show the characters, opinions and qualities of a group of users. Try to create 5 personas associated with your service.

What do they look like? Draw/create an image here.



Photo by Andrew Robinson on Unsplash

NAME:

Sam

AGE:

30

OCCUPATION:

B2B Salesman

LIKES:

Trying new things

DISLIKES:

Feeling left out

FAMILY AND FRIENDS:

Wife

WHAT WOULD THEY USE THIS SERVICE FOR?

Come up with an investment
strategy



WHAT ARE THEY MOST LIKELY TO SAY ABOUT THE SERVICE?

“

Looks promising

”

Figure 64: Sam Persona

PERSONA

Personas are used to capture the type of people involved in and using your service. They are not necessarily a representation of an individual, but show the characters, opinions and qualities of a group of users. Try to create 5 personas associated with your service.

What do they look like? Draw/create an image here.

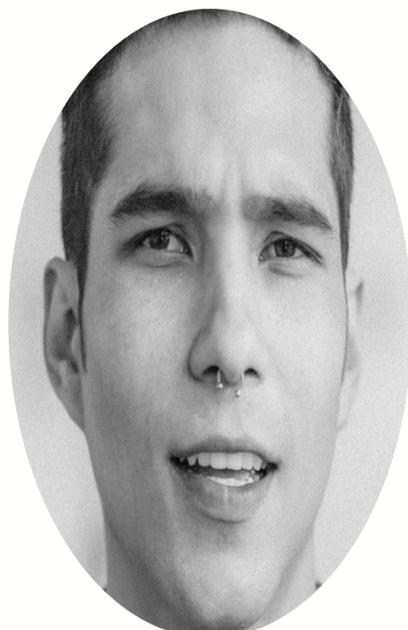


Photo by Roberto Delgado Webb on Unsplash

WHAT ARE THEY MOST LIKELY TO SAY ABOUT THE SERVICE?

“

they've got all the cryptos

”,

NAME:

Timmy

AGE:

18

OCCUPATION:

High school student

LIKES:

Cryptocurrencies

DISLIKES:

Studying

FAMILY AND FRIENDS:

Lives with parents

WHAT WOULD THEY USE THIS SERVICE FOR?

Compare cryptocurrencies



Figure 65: Timmy Persona

References

- [1] M. Aukia, A. Boehm, A.L. Heath, D. Joffe, G. Stoian, and M. Veiner. Thalia White Paper, 2019.
- [2] M. Aukia, A. Boehm, A.L. Heath, D. Joffe, G. Stoian, and M. Veiner. Thalia Technical Report, 2019.
- [3] PayScale. Average Software Developer Salary in United Kingdom. Available at: https://www.payscale.com/research/UK/Job=Software_Developer/Salary.
- [4] B. M. Ferreira, S. D. J. Barbosa, and T. Conte. PATHY: Using Empathy with Personas to Design Applications that Meet the Users' Needs, 2016.
- [5] Interaction Design Foundation. Ux Design Courses for Beginners and Professionals. Available at: <https://www.interaction-design.org/>.
- [6] R. F. Dam and Y. S. Teo. Personas – A Simple Introduction. Available at: <https://www.interaction-design.org/literature/article/personas-why-and-how-you-should-use-them>.
- [7] L. H. Shawn. Accessibility in User-Centered Design: Personas. Available at: <http://uiaccess.com/accessucd/personas.html>.
- [8] D. Bland. Agile Coaching Tip: What Is an Empathy Map? Available at: <https://www.solutionsiq.com/resource/blog-post/what-is-an-empathy-map/>.
- [9] Quantpedia. A comprehensive list of tools for quantitative traders. Available at: <https://quantpedia.com/links-tools/?category=historical-data>.
- [10] Fidelity. Fidelity Active Trader Pro. Available at: <https://www.fidelity.com/trading/advanced-trading-tools/active-trader-pro/overview-c>.
- [11] MetaTrader. MetaTrader4 Trading Platform. Available at: <https://www.metatrader4.com/en>.
- [12] NinjaTrader. NinjaTrader Platform. Available at: <https://ninjaTrader.com/>.
- [13] Portfolio Visualizer. Portfolio Visualizer Backtesting Tool. Available at: <https://www.portfoliovisualizer.com/backtest-portfolio>.
- [14] ETFReplay. Etfreplay Backtesting Tool. Available at: <https://www.etfreplay.com/combine.aspx>.
- [15] Backtest Curvo. Backest Curvo Backtesting Tool. Available at: <https://backtest.curvo.eu/>.
- [16] Stockbacktest. Stockbacktest Backtesting Tool. Available at: <http://www.stockbacktest.com/>.
- [17] J. Williams. The Basics of Branding. Available at: <https://www.entrepreneur.com/article/77408>.
- [18] R. Clifton. *Brands and Branding*. Economist Books, 2009.
- [19] Pantone. Pantone Color of the Year 2020. Available at: <https://www.pantone.com/color-intelligence/color-of-the-year/color-of-the-year-2020>.
- [20] Sessions College. Triadic Color Scheme Wheel. Available at: <https://www.sessions.edu/color-calculator/>.
- [21] San Jose State University. FURPS+ Model. Available at: <http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/ooa/requirements/IdentifyingURPS.htm>.
- [22] A. B. Al-Badareen, M. H. Selamat, M. A. Jabar, J. Din, and S. Turaev. Software Quality Models: A Comparative Study. *Communications in Computer and Information Science*, 2011.
- [23] R. Dhall. Designing Graceful Degradation in Software Systems. *Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering*, 10(-):171–179, 2017.

- [24] D. P. Drljaca and B. Latinovic. Using TELOS for the planning of the information system audit. *IOP Conference Series: Materials Science and Engineering*, 294:012022, 2018.
- [25] B. Olsen and L. Kolvereid. Development of new ventures over time: strategy, profitability and growth in new Scandinavian firms. *Entrepreneurship & Regional Development*, 6(4):357–370, 1994.
- [26] FCA. Understanding ‘advice’ and ‘guidance’ on investments, 2018. Available at: <https://www.fca.org.uk/consumers/understanding-advice-guidance-investments>.
- [27] NumFOCUS. pandas - Python Data Analysis Library. Available at: <https://pandas.pydata.org/>.
- [28] W. McKinney et al. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.
- [29] MDN Web Docs. HTML5 Developer Guides. Available at: <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>.
- [30] Bulma. Bulma: Free, Open Source, and Modern CSS Framework based on Flexbox. Available at: <https://bulma.io/>.
- [31] plotly. Introducing Dash. Available at: <https://medium.com/plotly/introducing-dash-5ecf7191b503>.
- [32] MDN Web Docs. Web Technology for Developers - JavaScript. Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [33] plotly. Dash Layout. Available at: <https://dash.plotly.com/layout>.
- [34] plotly. Dash DataTable. Available at: <https://dash.plotly.com/datatable>.
- [35] R. Brath and M. Peters. Dashboard Design: Why Design is important. *DM Review Online*, 2004.
- [36] A. Janes, a. Sillitti, and G. Succi. Effective Dashboard Design. *CUTTER IT JOURNAL*, 26, 2004.
- [37] Python Software Foundation. Decimal Fixed Point and Floating Point Arithmetic, 2020. Available at: <https://docs.python.org/3/library/decimal.html>.
- [38] S. v.d. Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.
- [39] apilayer. Fixer API Documentation. Available at: <https://fixer.io/documentation>.
- [40] T. Facen. Nomics API Documentation. Available at: <https://pypi.org/project/nomics-python/>.
- [41] R. Aroussi. Yahoo Finance API Wrapper Documentation. Available at: <https://pypi.org/project/yfinance/>.
- [42] Alpha Vantage. Alpha Vantage API Documentation. Available at: <https://www.alphavantage.co/documentation/>.
- [43] Quandl Inc. Quandl API Documentation. Available at: <https://www.quandl.com/>.
- [44] J. v. d. Bossche and K. Sheppard. Pandas Datareader Documentation. Available at: <https://pandas-datareader.readthedocs.io/en/latest/>.
- [45] The Internet Society. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Available at: <https://tools.ietf.org/html/rfc4180>.
- [46] Cisco Security. A Framework to Protect Data Through Segmentation. Available at: https://tools.cisco.com/security/center/resources/framework_segmentation.
- [47] C. Dye. Multiple Schema Versus Multiple Databases. Available at: <https://www.oreilly.com/library/view/oracle-distributed-systems/1565924320/ch01s04.html>.

- [48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [49] D. Thomas and A. Hunt. *The Pragmatic Programmer - 20th Anniversary Edition*. The Pragmatic Bookshelf, 2019.
- [50] S. Nanz and C. A. Furia. A Comparative Study of Programming Languages in Rosetta Code. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [51] A. G. Edge and W. Remus. The Impact of Hierarchical and Egalitarian Organization Structure on Group Decision Making and Attitudes. *Developments in Business Simulation & Experiential Learning*, 11, 1984.
- [52] H. Sousa. How to code review in a Pull Request. Available at: <https://blog.codacy.com/how-to-code-review-in-a-pull-request/>.
- [53] A. Cockburn and L. Williams. The Costs and Benefits of Pair Programming. Available at: <https://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF>, 2000.
- [54] J. Bancroft-Connors. Why physical task boards still matter. Available at: <https://www.leadingagile.com/2015/05/why-physical-task-boards-still-matter/>.
- [55] P. Smyth. Creating Web APIs with Python and Flask, 2018. Available at: <https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask>.
- [56] Pallets. Flask Template Documentation, 2010. Available at: <https://flask.palletsprojects.com/en/1.1.x/templates/>.
- [57] Flask-Login Team. Available at: <https://flask-login.readthedocs.io/en/latest/>.
- [58] WTForms Team. WTForms Documentation. Available at: <https://wtforms.readthedocs.io/en/stable/forms.html>.
- [59] plotly. Sharing data between Dash Callbacks. Available at: <https://dash.plot.ly/sharing-data-between-callbacks>.
- [60] plotly. Advanced Callbacks. Available at: <https://dash.plotly.com/advanced-callbacks>.
- [61] plotly. Dash Dev Tools. Available at: <https://dash.plotly.com/devtools>.
- [62] plotly Dash Community. Dash Dynamic Callbacks Possible Solution. Available at: <https://community.plotly.com/t/dynamic-controls-and-dynamic-output-components/5519/2>.
- [63] plotly. Sharing Data Between Callbacks. Available at: <https://dash.plotly.com/sharing-data-between-callbacks>.
- [64] plotly. Integrating Dash with Existing Web Apps. Available at: <https://dash.plot.ly/integrating-dash>.
- [65] J. Thomas. Bulma Documentation, 2020. Available at: <https://bulma.io/documentation/overview/responsiveness/>.
- [66] G. Becedillas. PyAlgoTrade Documentation. Available at: <https://github.com/gbeced/pyalgotrade>.
- [67] P. Morissette. bt Documentation. Available at: <https://pmorissette.github.io/bt/>.
- [68] UK Government. The Data Protection Act. Available at: <https://www.gov.uk/data-protection>.
- [69] Db-Engines. Database Engine Ranking. Available at: <https://db-engines.com/en/ranking>.
- [70] Datanyze. Databases Market Share Report. Available at: <https://www.datanyze.com/market-share/databases-272>.
- [71] A. M. Al Balushi. A Framework for Data Migration between Various Types of Relational Database Management Systems. *International Journal of Computer Applications*, 123(8):38–43, 2015.

- [72] S. Pearlman. Understanding Data Migration: Strategy and Best Practices. <https://www.talend.com/resources/understanding-data-migration-strategies-best-practices/>.
- [73] E. Mueller. What is DevOps? Available at: <https://theagileadmin.com/what-is-devops/>.
- [74] I. Stapleton. Flake8 Official Documentation. Available at: <https://flake8.pycqa.org/en/latest/manpage.html>.
- [75] L. Langa. Black Documentation. Available at: <https://black.readthedocs.io/en/stable/>.
- [76] Software Freedom Conservancy. githooks - Hooks used by Git. Available at: <https://git-scm.com/docs/githooks>.
- [77] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.
- [78] J. Holck and N. Jørgensen. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems*, Special Issue 2003/2004.
- [79] Docker Inc. What is a container? Available at: <https://www.docker.com/resources/what-container>.
- [80] Docker Inc. Docker Documentation. Available at: <https://docs.docker.com/>.
- [81] CircleCI. CircleCI Continuous Integration and Delivery. Available at: <https://circleci.com/>.
- [82] R. Connolly. *Fundamentals of Web development*. Pearson Education, 2015.
- [83] Heroku. Heroku Cloud Application Platform, 2020. Available at: <https://www.heroku.com/>.
- [84] Heroku. SQLite on Heroku, 2017. Available at: <https://devcenter.heroku.com/articles/sqlite3>.
- [85] Amazon Web Services. Aws, 2020. Available at: <https://aws.amazon.com/>.
- [86] CircleCI. CircleCI and AWS. Available at <https://circleci.com/integrations/aws/>.
- [87] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015.
- [88] C. Benoit. Gunicorn Documentation, 2020. Available at: <https://docs.gunicorn.org/en/stable/>.
- [89] Intland Software. Quality Assurance Using Code Coverage Analysis. Available at: <https://content.intland.com/blog/sdlc/quality-assurance-using-code-coverage-analysis>.
- [90] H. Krekel. pytest Documentation. Available at: <https://docs.pytest.org/>.
- [91] M. Foord. Mock Documentation. Available at: <https://mock.readthedocs.io/en/latest/>.
- [92] Software Freedom Conservancy. The Selenium Browser Automation Project. Available at: <https://www.selenium.dev/documentation/en/>.
- [93] N. Batchelder. Coverage Documentation. Available at: <https://coverage.readthedocs.io/>.
- [94] Software Testing Help. What is Test Data? Test Data Preparation Techniques with Example. Available at: <https://www.softwaretestinghelp.com/tips-to-design-test-data-before-executing-your-test-cases/>.
- [95] S W.. Ambler. Introduction to Test Driven Development. Available at: <http://agiledata.org/essays/tdd.html>.
- [96] W.-S. Tan, D. Liu, and R. Bishu. Web evaluation: Heuristic evaluation vs. user testing. *International Journal of Industrial Ergonomics*, 39(4):621–627, 2009.
- [97] SAFe. Release on Demand. Available at: <https://www.scaledagileframework.com/release-on-demand/>.

- [98] Amazon Web Services. Amazon Compute Service Level Agreement. Available at: <https://aws.amazon.com/compute/sla/>.
- [99] Google Developers. Tools for Web Developers - Lighthouse. Available at: <https://developers.google.com/web/tools/lighthouse>.
- [100] Nomics. Nomics Cryptocurrency Bitcoin API. Available at: <https://p.nomics.com/cryptocurrency-bitcoin-api>.
- [101] Financial Conduct Authority. Understanding ‘advice’ and ‘guidance’ on investments. Available at: <https://www.fca.org.uk/consumers/understanding-advice-guidance-investments>.
- [102] Portfolio Visualizer. Portfolio visualizer - Terms of Service. Available at: <https://www.portfoliovisualizer.com/terms-of-service>.
- [103] Trust Arc. *Essential Guide to the GDPR*. Trust Arc, 2018.
- [104] UK Government. Assistive technology tools you can test with at no cost. Available at: <https://accessibility.blog.gov.uk/2018/09/27/assistive-technology-tools-you-can-use-at-no-cost/>.
- [105] Inc. Stripe. Stripe Payment Platform. Available at: <https://stripe.com/>.