Thalia Technical Report

Team Charlie

Martti Aukia, Alber Boehm, Arthur-Louis Heath, George Stoian, Daniel Joffe, Marcell Veiner

University of Aberdeen
25.11.2019

Contents

1 Introduction

This report will present our findings and progress over the course of the semester. First, we will describe both the functional and non-functional requirements. We then move on to discussing the System Architecture. Next, we present the risks we have identified as non-negligible and the strategies we have developed for mitigating them. This will be followed by our Initial Project Plan, which defines our approach to creating the product and specifies a schedule for development. Finally, we will discuss the problems we have encountered and the solutions we have found in trying to solve them.

2 Requirements

We classify our requirements using the established FURPS+ model [1]. Below you will see 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 to provide a compelling product for paying customers.

2.1 Functional Requirements

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

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

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

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

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

2.2 Non-functional Requirements

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

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

5. Interfacing:
   - The Data Gathering Module must never use APIs stated to-be-deprecated within a month.
   - The Data Gathering Module must not exceed its contractual usage limits.
6. Operations:
   - An administrator on-call will be necessary for unexpected issues.
7. Packaging:
   - The product needs to work inside a Linux container (e.g. Docker).
   - All dependencies need to be installable with a single command.
8. Legal:
   - All user testing must be done with ethical approval from the University.
   - UI must display a clear legal disclaimer about the service not providing financial advice.
   - All third-party code should allow for commercial use without requiring source disclosure (e.g. no GPL-3).
   - User data handling should comply with GDPR.

3 Architecture Choice

We started by looking at the main architecture types [3] and checking their advantages and disadvantages with respect to the specifics of our application. This has helped us greatly reduce the number of candidate architectures. The remaining ones specific to web development were *Client-Server*, *Data-Centric* and *Layered* types [4]. The options that proved to be the most advantageous were the *Hexagonal* and the *Three Layer* architectures. However, due to the fact that a large part of our business logic revolves around data collection and processing, it was decided that steps should be taken in order to isolate the financial data management. As a result, we have modified the *Hexagonal* architecture to include a marginal *Financial Data Manager*. For the *Three Layer* architecture, we have added an extra layer below the data storage layer. After comparing the two we have decided that the modified *Three Layer* architecture represented our application well while offering more simplicity than the *Hexagonal* architecture.

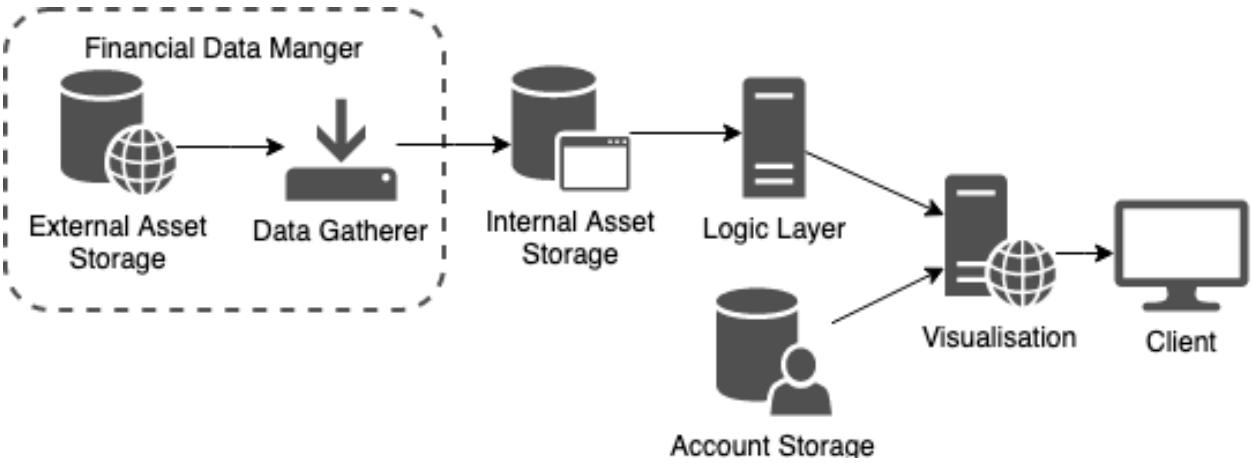


Figure 1: Architecture Diagram

3.1 Justifying a Financial Data Retrieval layer

The first argument comes from the need for extra security (guidelines have been obtained from the National Cyber Security Centre for Separation and cloud security [5].) By giving the web-server limited permissions for accessing the financial data, we limit the type of attacks that can take place. Not including the Data Gathering Module in the business logic core also allows for higher availability since the web-sever and the data retrieval can work independently and thus also be modified independently. Finally, the calls to third-party financial data APIs require private keys that we want to keep in a more secure manner than having the web-server have access to them.

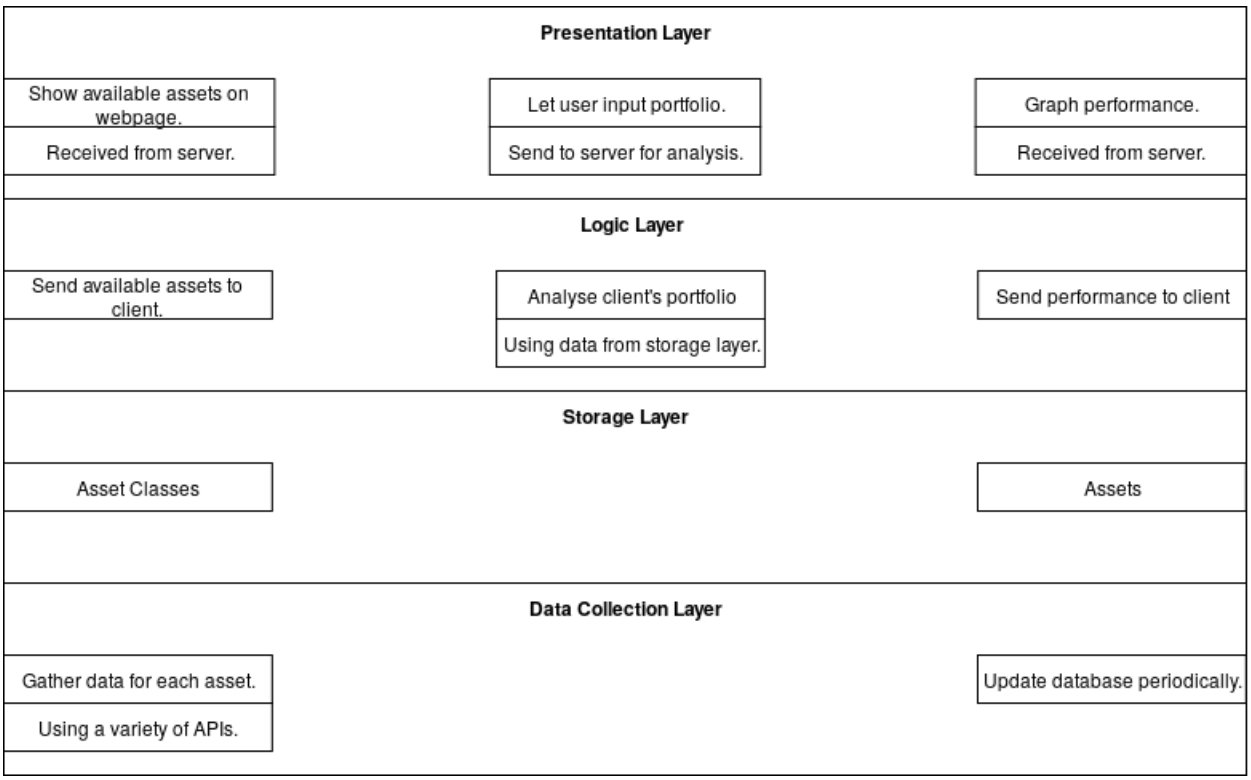


Figure 2: Layer View of Architecture

3.2 Portfolio Data Representation

We have chosen to create two separate data types for a portfolio - one for its specification (how much to invest in what) and one for its performance (how much money one has at a given moment in time). The client constructs the specification based on user input and sends it to the server. Once received, the server does the required computations and sends back the portfolio performance to the client. The drawback is that the response time might be increased when doing intensive numerical calculations for multiple clients, but the benefits are the following:

- The client does not need to perform expensive computations on their machine.
- Letting the client access as little information as possible (e.g. preventing access to business logic code) is more secure.
- Minimal information is passed over networks, which can be slow and unreliable.

3.3 Data Harvester

The Data Harvester is a generic adapter for third party APIs offering financial data. Hence, it allows for configuration of the API endpoint, the maximum requests per minute/hour/day and the parser required for formatting the received data. Every module is connected to the database and does some light parsing on the data it receives. It standardizes the collected data so that it can be saved in an SQL database. The standardization can be done in two ways. If supported by the API, the data is pulled into a Pandas data frame [6], cleaned from unnecessary information, converted to CSV and inserted into the database. Otherwise, it is parsed as text, cleaned from unwanted values, placed in CSV format and inserted into the database.

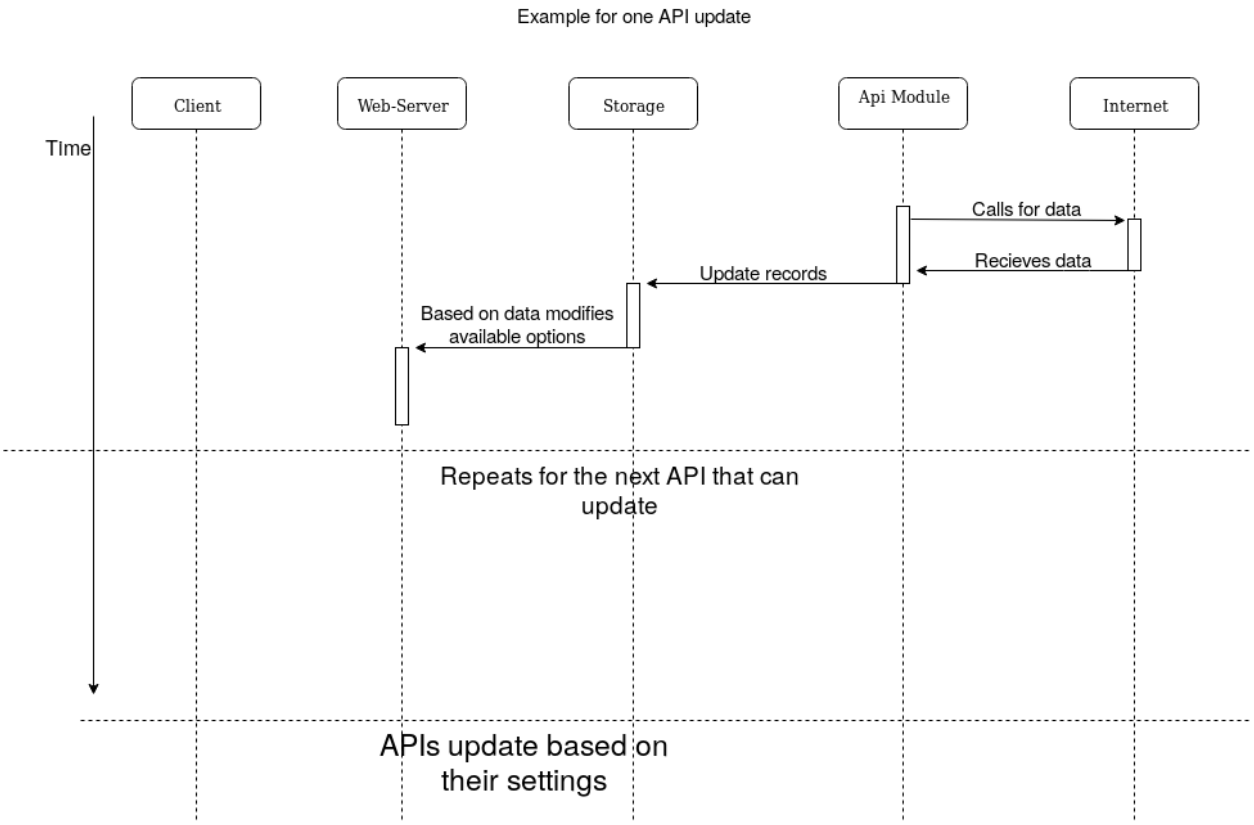


Figure 3: Time Events Diagram for Data Harvester

3.4 Data Usage Legalities

We have consulted the Terms and Conditions for the financial data APIs we use. After conducting this research, we have observed that for the APIs we are using no legal problems could arise.

3.5 Choice of Framework

We have used the Django framework without its Object-Relational Mapping (ORM) in order to have complete control over our data processing. The decision to use a Python framework has been made as a matter of familiarity with the programming language. The two popular frameworks, Django and Flask, exhibit different philosophies. The former provides you with a variety of built-in functionality (e.g. Authentication, Prevention of SQL Injection) at a cost of less flexibility in the development process. A more minimal framework, Flask sets its priorities the other way around. Since the complexity of our application lies within the business logic (which is fully framework-agnostic), we have decided to use Django over Flask to make the development of non-business logic components as simple as possible.

3.6 Choice of Frontend Technologies

The wireframe diagram below is a display of how our main page will look like. This wireframe model will be implemented using bootstrap-like libraries in order to speed up the process. The base of our frontend will be composed of HTML5, CSS3, and JavaScript which handle content, style, and functionality, respectively. We have chosen these technologies for the greatest browser compatibility to avoid restricting us in the future development of the frontend. In addition to using JavaScript, we will be using the plot.ly library for creating attractive graphs, which strikes the right balance between functionality and complexity for our needs.

Figure 4: UI wireframes

The following diagram contains a typical user interaction with our website.

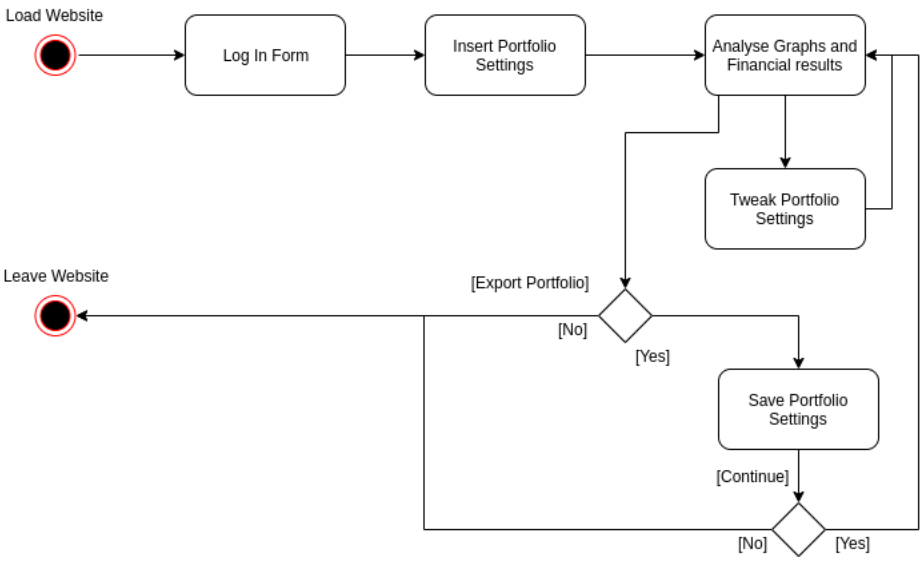


Figure 5: Activity Diagram

3.7 Inter-layer Communication

Our Inter-Layer communication has been based on the principle of *Separation of Concerns* [7]. Following this principle, we have made sure that at each step each layer knows only as much is it needs. The style and presentation are sent to the client machine, upon which only the client is able to interact with it. The Business Logic has been placed as a separate module on the web-server and it only has read access to the financial data in the database. The logic of the Data Harvester is outside of the Django Web-Server and it does not have any connection to any of the website components.

The following diagram illustrates the full backtesting cycle from the viewpoint of communication between components.

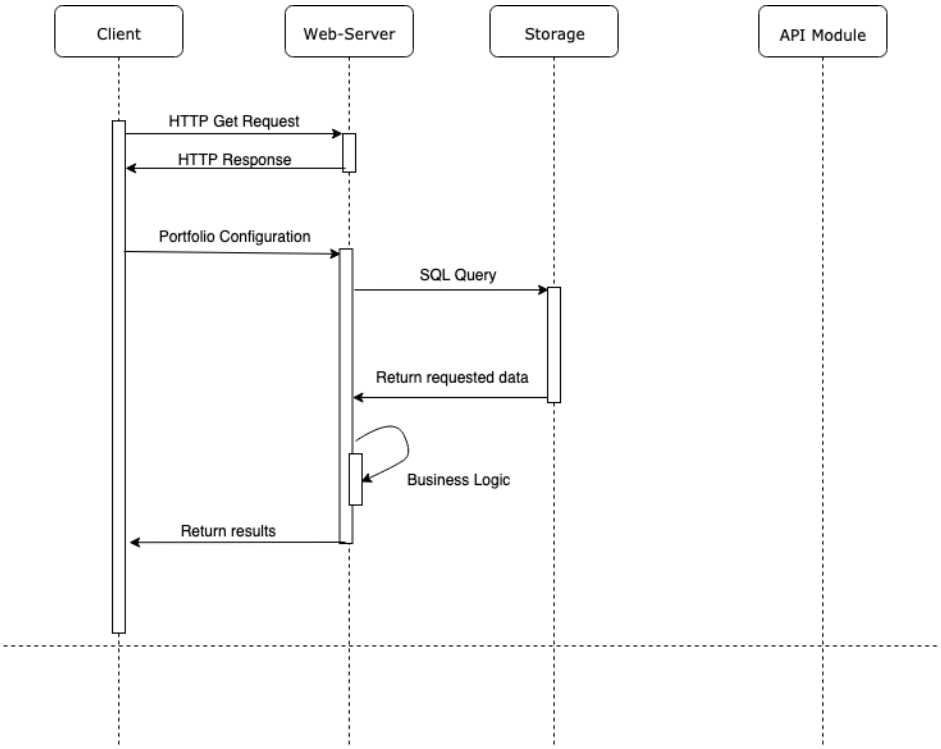


Figure 6: User Interaction Process Diagram

4 Initial Risk Assessment

The following section will highlight the risks that may affect the successful delivery of the software product. We have analyzed the impact of all risks across three dimensions - likelihood, severity, and detectability. The result of this analysis can be summarised in the following risk assessment matrix.

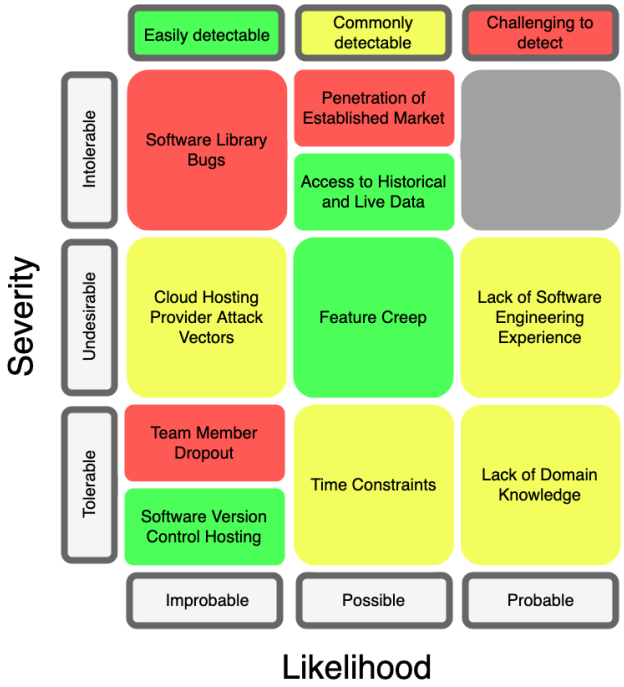


Figure 7: 3D Risk Matrix

For additional details and insight into our mitigation strategies, we will cover each risk from most to least impactful.

4.1 Penetration of Established Market

Backtesting is an established practice among both professional and retail investors. We have to convince people of the superiority of our product. It is unlikely that we will convince investors who have purchased a lifetime license for some backtesting tool to switch to our platform. This is why we consider subscription-based tools as our direct competition. Additionally, we believe that competing with backtesting tools developed for institutional investors will be difficult due to our budget constraints. Hence, we are focusing on beating competitors within the market of subscription-based tools for retail investors. To gain market share, we have identified three key factors we can leverage to build a superior solution. These are:

- Price - We aim to undercut competitors to make switching that much easier.
- UI - Existing tools often look very unappealing. We aim for creating an enjoyable user experience.
- Feature Set - Solutions on the market are either too expensive or limited in functionality. We want to offer an extensive suite of features that attracts investors from different backgrounds. Metrics for evaluating whether we have delivered on all of the above can be found in the 'Evaluation Strategy' section.

4.2 Access to Historical and Live Data

High-quality financial data is an expensive good - entire businesses are built on access to certain datasets. Additionally, it is much easier to find price data for some asset classes (e.g. equities) than others (e.g. commodities). Fortunately, our product does not require access to expensive datasets for initial deployment. Investing operates on much larger timeframes than trading and thus backtesting for investment purposes does not require extremely granular price data (daily is usually sufficient). This means that we can rely on publicly available datasets and APIs. To accommodate for rate restrictions on calls to the latter, we have made our API adapters highly configurable (see the 'Architecture Choice' section for a description of our

Data Gathering Module) such that we would never run into a situation where we exhausted our available calls per day. Moreover, we are able to aggregate price data for multiple sources for the same asset into a single data point in our asset price database. This reduces the effect of noisy data that may be a result of poor data gathering practices by the data providers. As for the availability of commodity price data, we are able to cover the majority of commodities with free data. For those that remain, we are actively exploring a subscription-based model with multiple data providers. Relying on APIs for providing live data makes us reliant on the uptime of these APIs. If any of them were to stop providing their services, we would have to find a replacement quickly to minimize downtime of our own platform. Thus, as a precaution, we are using at least two and up to four (where available) data providers for every asset live feed. This redundancy will keep our operations running in case one of them stops servicing our requests and gives us enough time to find a replacement.

4.3 Software Library Bugs

Our product relies on multiple popular open-source libraries and frameworks. Although these have been vetted and stress-tested by thousands of developers, there is a non-negligible chance that one of these libraries contains an undiscovered bug. Since our customers rely on the integrity of our data and calculations, a dysfunctional library could potentially lead them to a wrong conclusion about a given portfolio. In the worst-case scenario, this might be a decision to invest in assets that are considerably more risky than the customer believes them to be. Library bugs can be extremely challenging to detect. To mitigate this risk, we have developed a testing strategy that guarantees that our calculation results meet our expectations. Finally, it is worth reiterating that we are not providing financial advice and warn our customers to consult with a professional before making any investment decisions as part of our terms and conditions.

4.4 Lack of Software Engineering Experience

While some of our team members have worked on large software projects as part of an internship, most have little experience in developing a full-fledged software product. Therefore, there is no evidence that we will be able to handle the complexity involved in building Thalia. A challenge of detecting this risk is that it becomes apparent only as the development of the product progresses and complexity increases. To combat this risk, we are hosting regular retrospective meetings [8]. These give each team member the chance to go over issues they have run into during the last sprint. This allows us to support those struggling in the development process by providing additional training or assisting them through pair programming [9].

4.5 Feature Creep

The number of additional features that could be implemented on top of the core functionality is very large (see Appendix A for a full list of all optional features), which risks both overcomplicating the product and shifting the focus from core functionalities to extensions. We have identified two key strategies for limiting feature creep: preventing a feature with low utility from being developed and pruning features that are rarely used. The former can be implemented by asking our customers for features they would like to see added to Thalia. Commonly requested features would then be discussed by the team with a vote serving as a decision mechanism for whether it should be implemented. Decisions on which features to prune are a matter of collecting usage statistics. For example, a customer will have to activate the display of a non-standard performance metric via the settings menu (an action we can track in anonymized fashion). This allows us to gain insight into the popularity of any given feature after it has been deployed.

4.6 Cloud Hosting Provider Attack Vectors

Hosting our application on a machine that is not controlled by us (i.e. in the cloud) exposes us to security risks. These are often hard to estimate, as is evident by several security breaches that have occurred in the past [10]. Additionally, we have to place our trust in the hosting provider to deliver on the promised uptime guarantees. Since we are not processing payments ourselves, the types of sensitive data contained in our system are restricted to personal data of users (excluding payment information) and API access

keys. Security concerns involving the latter are covered below ('Software Version Control Hosting'). To reduce the attack surface, the former is stored in a database which, on top of being secured by a password, accepts connections only from a preconfigured set of IP addresses (the web-facing Django server and our Data Harvester system). Cloud Hosting is a highly competitive business and the cost of changing provider in case of missed uptime guarantees is low. Thus, we will monitor the uptime of our systems and switch provider in case it is necessary.

4.7 Lack of Domain Knowledge

The majority of team members have had little exposure to the domain knowledge required for building parts of our product. A misunderstanding could lead to incorrect implementation of a feature which would put us at risk of being unable to keep up with our schedule. This risk is entirely internal to our operations and thus more easily controlled. To combat the isolation of knowledge in some members, we have hosted workshops that help spread it to those with little experience in financial markets. Additionally, each member has done extensive reading on topics that concern backtesting in a non-trivial way. Lastly, our regular retrospective meetings require each team member to report their progress over the last sprint. Potential issues concerning misunderstandings or confusion over domain knowledge can thus be spotted early enough to avoid any negative knock-on effects.

4.8 Time Constraints

As for any project, tight deadlines can negatively affect the quality of the end result [11]. Since this product is developed within the scope of a university course, the time each member can allocate to its development is affected by a variety of external factors. Consequently, we have based our schedule and any estimates on a conservative amount of time each member has to commit to the project to allow for successful delivery (8 hours per week). This gives us confidence in being able to deliver a working product that passes our quality checks in the evaluation stage. If required, we may adjust the schedule by reducing the time spent on developing optional features towards the end of the semester and prioritize core deliverables instead. Hence, we are minimizing the likelihood of having to rush development in fear of being unable to meet the deadline otherwise.

4.9 Team Member Dropout

As reported by our supervisor, it is possible (although unlikely) that members of the team drop out of the course due to unforeseen circumstances. A reduced headcount puts us at risk of being unable to meet the requirements and could result in loss of product knowledge if it were to be isolated in the person who dropped out. Preventing this issue altogether is impossible. However, you can minimize the damage it may cause to the development of the project. The Egalitarian Team Structure maximizes our 'truck factor', which is a measure of the number of people who would have to stop working on the project to cause development to stall [12]. Every team member is familiar with the code underlying other parts of the product which are outside the scope of his weekly development efforts. Nevertheless, we would have to organize emergency meetings to discuss the impact of such a situation as it arises.

4.10 Software Version Control Hosting

Security breaches involving SVN hosting providers such as GitHub are not unheard of [13]. Commonly, these are the result of storing sensitive information (such as access keys) in a public repository. As mentioned previously, we have decided to treat the Data Processing Module (Data Harvester) as a separate component in our system which none of the other components is able to access. Additionally, we are storing API keys as environment variables in a file that isn't tracked by our SVC system. This minimizes the probability of us exposing sensitive data.

5 Initial Project Plan

5.1 Team Organisation

Our workflow is centered around the GitHub platform and the tools it provides, such as a ticketing system, pull requests and a scrum board for ongoing tasks. Our goal is not to have fixed responsibilities in our team so that everybody at some point will be required to develop a feature for each part of the system. Studies have also shown that any sort of status difference within a team can distort the error-correcting mechanism [14]. Because of these reasons we have decided to commit ourselves to the Egalitarian Team structure, which will introduce the required amount of flexibility into our team. To measure productivity, we use effort-oriented metrics, and assign story points to each ticket, based on the time needed and the functionality of the task. This is done by the team member that the ticket was assigned to, and is later reviewed by another member. As in many Agile teams, story points are based on numbers of the Fibonacci sequence, which forces us to consider the value of each ticket carefully and motivates splitting it into two separate tasks if necessary [15].

5.2 Evaluation Strategy

We also plan to make use of the tools offered by GitHub for testing and reviewing. Each pull request is reviewed by other team members before any changes are pushed to the production system. CI scripts are also in place to ensure code quality and integrity. This is done by enforcing the use of Pytest, and a style-checker (either flake-8 or black). We will also heavily depend on continuous user testing throughout development [16]. Our plan is to obtain the necessary ethical approval from the university before the start of development. Continuous user testing is our main part of the evaluation strategy, and as such it will help us to develop features unlike or better than existing ones. This is of key importance as we are about to enter an established market and we would like our product to be as distinguishable as possible.

5.3 Budget

As we are developing this product as part of a university module, we do not have any budget restrictions other than time. All of us are committed to allocating a minimum of 10 hours per week to development efforts and are willing to go beyond that if needed. In order to ensure that we are up to schedule, weekly meetings will be held, which allow the adjustment of workloads.

5.4 Milestones

Although Agile methods offer a great amount of flexibility, they do require some sort of governance [17]. For this reason, we introduced some important milestones, which will help us to stay on track. We have identified three milestones in the development process, with the last one being much more open-ended than the first two. These are the following:

5.4.1 Minimum Viable Product

This version enables an investor to create a portfolio from a core set of assets and plot its performance versus a predetermined indexing strategy. Since this is not a prototype but a functional product, all future development will expand on this codebase by adding additional features.

5.4.2 Thalia Release 1.0

As defined in our whitepaper, we strive to support five key use cases upon the public release of Thalia. Hence, this requires the following features to be fully functional:

- Creating a portfolio from a large set of assets drawn from the major asset classes using input forms.
- Weighing assets relative to each other by assigning a percentage to them using a range input.
- Plotting of a portfolio's performance over a time-series upon user request.

- Selection of one of the multiple indexing strategies and lazy portfolios from a drop-down list.
- Display of key metrics in a table.
- Specification of regular contributions and selection of a rebalancing interval through input forms.

We consider reaching this milestone as sufficient for the scope of this course.

5.5 Development of Additional Features

If time permits and development goes as planned, we have an array of additional features that we would like to see included in our product. For an overview, please have a look at Appendix A.

5.6 Schedule

Long-term planning is just as important in Agile development as in alternative planning procedures [18]. As this term we had a chance to set up the working environment and measure the velocity of development, we can make a prediction of the schedule. During the second semester, we will have a total of 12 weeks for developing the product. As we are following the Agile approach, we have divided the schedule according to milestones. Each one of these periods will include the development of some or multiple features, as well as testing and deployment, leaving us with a fully functional product. The scope for each milestone is defined as follows:

- **Week 0 - 4: MVP / Closed Alpha**
  - Key Metrics
  - Integration of Historical Datasets
  - Additional Lazy Portfolios (Benchmarks)
- **Week 4 - 8: Open Beta**
  - Creating User Accounts
  - Integration of Live Data APIs
  - Regular Contribution and Rebalancing
- **Week 8 - 12: Public Release**
  - Exporting Portfolio to PDF
  - Additional Features

As previously stated, we aim at including a number of additional features in order to better distinguish our product. The following is a product roadmap based on this schedule:

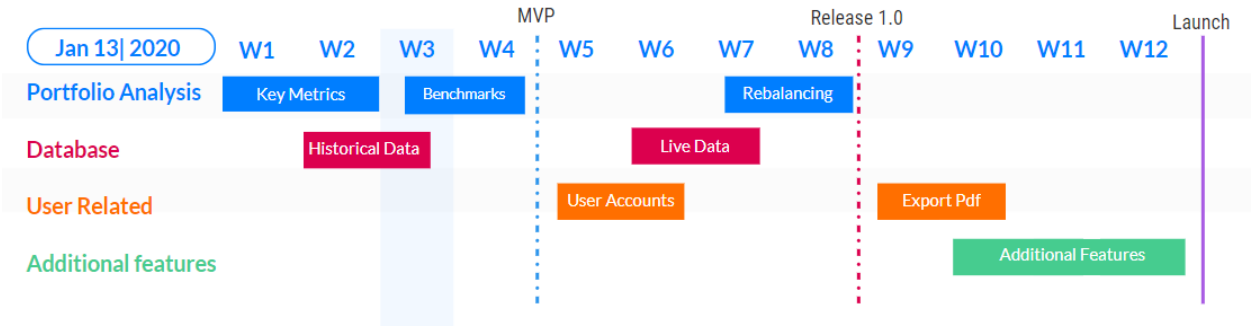


Figure 8: Roadmap

6 Testing Strategy

6.1 Scope

Our biggest concern is ensuring that the results we show to our users are correct. Outlined below are the main functionalities we need to be especially careful about when we test.

1. Results of key metric functions, e.g. Sharpe Ratio
2. Data plotting results
3. Handling of third-party API data
4. User authentication

Closer to the deployment date, we need to also test if the application works in a production environment. To minimize the "works on my machine"-effect, we will be using a container such as Docker to reduce differences between different development and production environments [19]. This also eases deployment to many of the cloud hosting providers (e.g. Amazon Web Services or Microsoft Azure) since they support hosting containers natively [20] [21].

6.2 Not in Scope

We assume our third-party libraries will work as advertised and as such we will not be testing them explicitly. Testing them would cause a large amount of overhead and would make the project impossible to achieve in our allocated time frame. To alleviate this, we will only use mature libraries with large communities and develop our own code when this is not possible.

6.3 Strategy in Brief

For most of our testing, just employing good testing practices will be enough. All public APIs must have unit tests, all interfacing modules must have integration tests and the most critical code must have extensive tests for different branching paths and boundary checking. No assumptions should be made about user input since the nature of our application demands rigorous verification of intent. Input should be verified to be acceptable for the system and any problematic portfolios should be returned as erroneous with an appropriate error message to the user. Another potentially erroneous external input could come from our live data third-party APIs. Although we do carefully select our APIs, we should not blindly trust them. We have the convenience of having alternative sources for live data, hence we can verify our primary data sources' output against others. As our live data are updated daily, the additional overhead of verifying our data will not affect the performance of the UX.

7 Proof of Concept

7.1 Our Journey

Our primary challenge for this project has so far been training all the members for next spring's development. We wanted to avoid any one person being fully responsible for any single task, which is in line with our egalitarian team philosophy. This became even more important as we noticed that our 'truck factor' was one since only one of our team members had any substantial experience with financial markets. The biggest skill gaps that we identified were familiarity with finance, web development experience and lack of experience with professional software development practices in general. During the fall, we spent a lot of time learning about backtesting and Agile processes in the aims of making the spring term development easier. Specifically, we have had a lot of workshops and assigned people tasks in areas they weren't familiar with before. This did mean that we spent quite a bit of time reviewing and teaching, but in the effort of having the whole team participate in the design and review process, this was highly beneficial.

7.2 Achievements

For the proof of concept, we wanted to test our key features. This required us to develop the Data Gathering Module, implement a subset of the metric algorithms and setting up a Django web-app. The proof of concept is a basic Django app that runs on localhost, uses some financial data, can calculate some of the more important backtesting metrics and displays a graph of the portfolio's performance for the past 2 years. Something else worth mentioning is the success in adopting our Agile process. We use scrum style sprints [22], which in combination with our continuous integration practices have made the process of creating work tasks, dividing them, and reviewing them relatively painless.

7.3 Problems

The biggest problem we faced was getting accustomed to working as a new team. Another problem has been evaluating how much work we can get done. Calculating work velocity has sometimes been difficult as most of our efforts have devoted to initial design and learning.

7.4 Things that Work as Expected

Since all of us have extensive experience with the Python programming language, getting the proof-of-concept Django application up and running was not a great challenge. The functionalities offered by Django work as expected and we have had no issues in creating the basic building blocks (i.e. Data Gathering Module, SQLite database, and our web-server). Over the course of the semester, we have used the GitHub issue tracker and ticketing system to organize our work. Having both systems on the same platform has been of great use to us since it is easy to reference issues within tickets and vice versa. We expected a ticketing system to help us structure our development process and found that to be true throughout the course. Finally, the use of Continuous Integration scripts has ensured the coherence of our codebase and allowed us to verify whether any commit would disrupt existing functionality before merging it.

7.5 Changes to be made

As we are moving towards hosting our service, we will 'containerize' (using Docker, Vagrant, or the like) our application to avoid issues with deploying to a machine different from the one used for development [20][21].

8 Conclusion

In this report, we presented our findings in developing a backtesting service. We have shown how our architecture choices serve both our functional and non-functional requirements. As part of this, we outlined the specification for our Data Gathering Module and shown how it integrates into a traditional Three Tier Architecture. Moving forward, we are implementing the measures identified in the risk assessment to minimize the probability of failure. Together with reaching the milestones defined in our roadmap, this will allow us to deliver a fully-functional product that is secure for our customers to use.

9 Appendices

9.1 Appendix A - Optional Features

- Choice of currency for displaying absolute values
- Receive warnings on backtest which is overfitted to a timeframe
- The system must not itself handle payments
- Use leverage
- Set upper and lower bound on asset price at which to buy or sell a specific asset

- Warn user about lack of historical data or short timeframe selected by them
- Flagging market-moving events
- Support for Technical Analysis Patterns
- Integration of a Scripting Language to simulate automated trading strategies

9.2 Appendix B - Glossary

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

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

9.3 Appendix C - Proof of Concept Use Case

After initially connecting to the website, click "Let's go" to begin using Thalia.

Figure 9: Home Page

Thalia currently only supports 3 indices - the Dow Jones, S&P500, and the NASDAQ. Select these from the drop-down menus in any order.

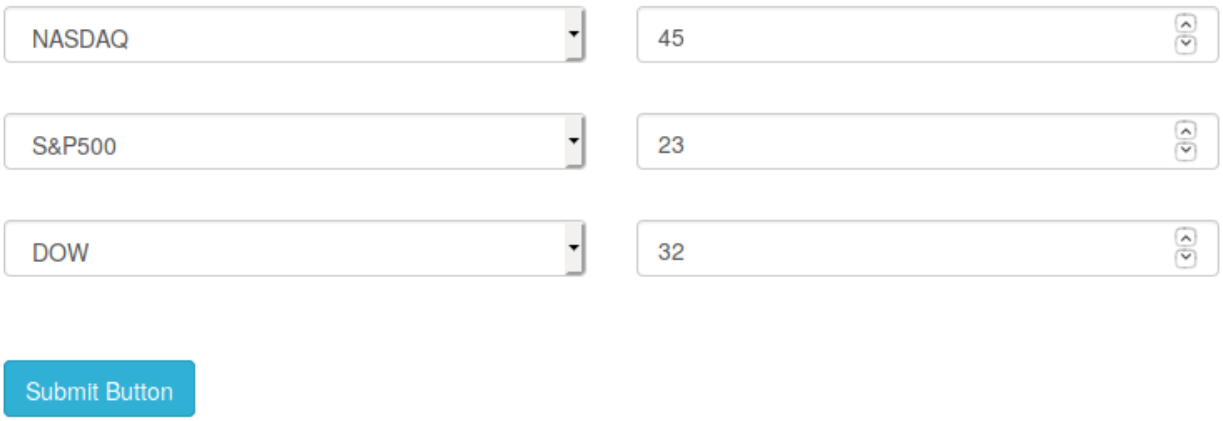
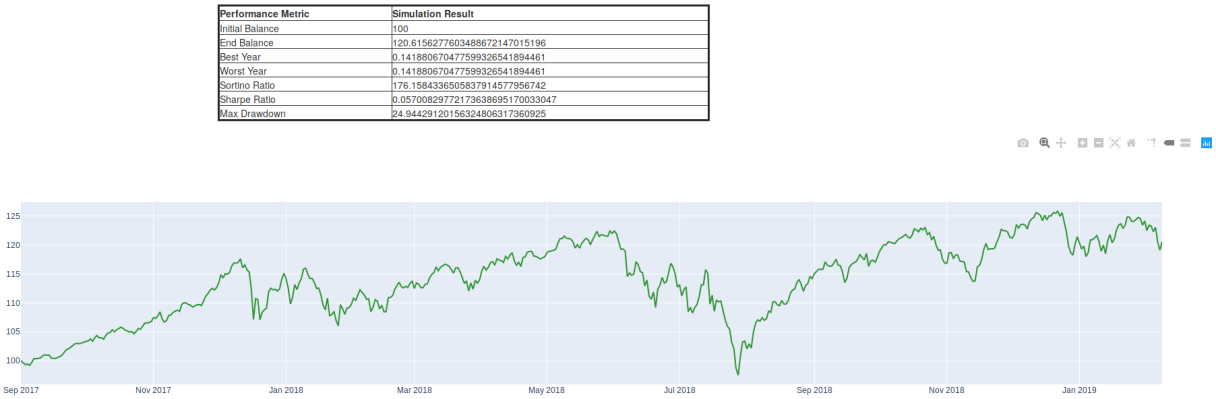


Figure 10: Allocation Page

Thalia doesn't yet support absolute currency values, so enter your investment in each as a percentage without using the '%' sign. If you don't want to invest at all in an asset, enter 0. When you're done, click "Submit Button". This will take you to the results page, where you can see how your portfolio has performed over the past couple of years.



Performance Metric	Simulation Result
Initial Balance	100
End Balance	120.61562776034886721470151 96
Best Year	0.14188067047759932654189446 1
Worst Year	0.14188067047759932654189446 1
Sortino Ratio	176.15843365058379145779567 42
Sharpe Ratio	0.05700829772173638695170033 047
Max Drawdown	24.94429120156324806317360925

Figure 11: Results Page

All financial results are given as a percentage of your initial investment - an end balance of 120 means your portfolio's value has increased by 20%. There are two parts to the results page - a dashboard of key figures and a graph. The definition of the key figures can be found in Appendix B. The graph is a more complete view of your portfolio's value at specific points in time.

Performance Metric	Simulation Result
Initial Balance	100
End Balance	120.61562776034886721470151 96
Best Year	0.14188067047759932654189446 1
Worst Year	0.14188067047759932654189446 1
Sortino Ratio	176.15843365058379145779567 42
Sharpe Ratio	0.05700829772173638695170033 047
Max Drawdown	24.94429120156324806317360925

Figure 12: Results Page

Note that all the assets Thalia currently supports are highly correlated, so you can expect the graph and dashboard to look fairly similar for different portfolios.

10 References

[1] [FURPS+]http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/ooa/requirements/IdentifyingURPS.htm

[2] [Loading times for mobile users][Google Data, Global, n=3,700 aggregated, anonymized Google Analytics data from a sample of mWeb sites opted into sharing benchmark data, March 2016.]

[3] [Architecture types]IEEE Software 2006 Vol. 23 Issue No. 02 -March/April. Particularly Software Architecture-Centric Methods and Agile Development

[4] [Web Service Architectures]https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

[5] [Security guidelines]https://www.ncsc.gov.uk/guidance/separation-and-cloud-security

[6] [Pandas][https://pandas.pydata.org/]https://pandas.pydata.org/

[7] [Separation of Concerns]http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223

[8] [Agile Retros][https://www.atlassian.com/team-playbook/plays/retrospective

[9] [Pair Programming]https://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf

[10] [Cloud Hosting Security]https://techcrunch.com/2019/10/21/nordvpn-confirms-it-was-hacked/

[11] [Adding Programmers to a late project makes it later]The Mythical Man-Month, Fred Brooks, 1975

[12] [Truck Factor]http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/

[13] [GitHub Security Breach]https://techcrunch.com/2017/11/21/uber-data-breach-from-2016-affected-57-million-riders-and-drivers/

[14] [Team Hierarchies]https://absel-ojs-ttu.tdl.org/absel/index.php/absel/article/view/2208

[15] [Fibonacci Storypoint Values]https://www.atlassian.com/agile/project-management/estimation

[16] [User Testing] https://www.system-concepts.com/insights/tips-for-integrating-user-testing-into-an-agile-development-process/

[17] [Governance in Agile] https://www.agilest.org/agile-project-management/governance/

[18] [Agile Roadmaps] https://www.atlassian.com/agile/product-management/roadmaps

[19] [Lightweight Linux Containers] https://www.researchgate.net/publication/261960832_Docker_lightweight_Linuxq_containers_for_consistent_development_and_deployment

[20] [Hosting Docker Containers] https://aws.amazon.com/docker/#Run_Docker_on_AWS

[21] [Docker on Azure] https://azure.microsoft.com/en-gb/services/kubernetes-service/docker/

[22] [Scrum Sprints] [https://www.scrumguides.org/scrum-guide.html#events-sprint]