Quant Model Tools: A C++ Framework for Building, Testing, and Optimizing Trading Models

By

Yussof Kazmi, Bachelor of Science in Computer Science

A thesis submitted to the Graduate Committee of

Ramapo College of New Jersey in partial fulfillment

of the requirements for the degree of

Master of Science in Data Science

Spring, 2025

Committee Members:

Dr. Victor Miller, Advisor

Dr. Ali Al-Juboori, Reader

Dr. Lawrence D'Antonio, Reader

COPYRIGHT

© Yussof Kazmi

2025

Dedication

To all of the people who entered algorithmic trading and have created software tools to help build and test trading models, educate the algorithmic trading community, and provide an overall welcoming space, thank you and keep pushing the boundaries of quantitative finance.

Acknowledgments

I express my sincere gratitude to Dr. Victor Miller, Dr. Ali Al-Juboori, and Dr. Lawrence D'Antonio for their guidance and help throughout this entire process.

I would also like to thank my family, brothers, and my friends: Joseph Altamura, Tyler Hyde, Gagan Namburi, Nicholas D'Mato, Nomi Ahmed, and Matyas Alcivar for their support throughout this entire process.

I would like to thank Juan Navas and Xavier Chavez for their help and support during and after my time at EarthCam.

I would also like to thank James H. Michalski, John P. Carreiro, and Paul Fleming for introducing me to the world of Wall Street, Algorithmic Trading, and JFMX3X. Without their help, support, and willingness to give me a chance, none of this would have been possible.

Table of Contents

List of Tables

List of Figures

Abstract

Investing in financial instruments, such as stocks, has been a significant pillar of stability, savings, and wealth generation for decades due to the potential for positive returns. In response to such high investment activity, the stock market has risen exponentially over the last four decades, despite facing challenges such as severe market corrections, financial market crashes, and recessions. Over the years, this has led to maximizing positive investment returns while minimizing negative ones which has led to the emergence of algorithmic trading models. As trading models become more sophisticated, they require a substantial amount of analysis, including testing the model under simulated conditions, comparing trades, cleaning and sourcing price data, and optimizing profitable parameters.

This thesis presents a framework developed in the C++ programming language that enables the execution of multiple trading simulations using user-provided trading models, thereby generating meaningful performance insights for the user-provided model. This framework will allow users to configure their trading model through configuration files, backtest and optimize it across various simulated market conditions, and run WFA (Walk-Forward Analysis), which simulates a trading model against unseen live market data as if it were trading live in the market. Furthermore, the trading model will be able to utilize a wide variety of socks in its portfolio per the user's request.

The goal of this thesis is to develop the tools to combat overfitting in trading models. As the number of investors, the amount of money, and the complexity of trading activity in the market increases, there is a significant need for a tool such as this framework in developing robust trading models.

Chapter 1 Introduction

1.1 History of Trading and Markets

Trading financial instruments has been around for much longer than normally known, dating back to 12th-century France, where there was the managing and regulation of debts from agricultural entities on behalf of the banks by brokers. Since then, society has seen many different types of asset classes take center stage, and thus, more and more markets have been developed and created around the world. Each market would have a series of financial instruments/assets that could then be bet on or against, and eventually, shares of said instruments would then be allocated to various buyers from brokers.

Eventually, market exchanges such as the NYSE (New York Stock Exchange), would be founded on May 17th, 1792, and would allow stocks to be traded with payouts established per an original set of rules signed by 24 stock brokers at the time (NYSE, n.d.). This agreement would be known as the Buttonwood Agreement and the goal was to promote investing as a way of combating financial panic, provide the public confidence in the stock markets, and ensure trusted and sound investment deals were done between parties properly. This then led to Wall Street, which was already developed at the time to become the United State's financial capital in "The Compromise of 1790" (NYSE, 2014.).

In the last 100 years, there have been numerous stock market crashes that were very severe and impactful. Looking at examples such as "1929's stock market crash, dubbed Black Thursday", where the Dow Jones Industrial Average lost over 50% of its value in a matter of weeks (Richardson et al., 2013); or, "Black Monday" which was a crash that started in the Hong

Kong markets and eventually spread, impacting other markets around the world (Bernhardt & Eckblad, 2013). The "Black Monday" crash, at the end of October in 1987, had the following % drops in value for the following countries' stock markets: Hong Kong fell by 45%, Australia fell by 41.8%, the United Kingdom fell by 26%, and the United States fell by 22.68%.

However, some of the more recent crashes that are consistently referred to are the 2007-2009 Financial Collapse and the 2020 pandemic market crash. The 2007-2009 financial collapse was a period of recession where the stock market and other markets such as the housing and lending markets collapsed dramatically to the point where the S&P 500 dropped more than 50% and would not recover to those same levels until years later in 2013-2014 (Gratton, 2024). The 2020 pandemic crash was also a very sudden drop that happened on February 20, 2020, and lasted until April (Mazur et al., 2021).

Despite the crashes, corrections, and financial collapse, trading and markets of financial instruments have continued, but more importantly, evolved tremendously. Some of these evolutions in trading are an increase in the number of financial instruments for people to invest in, the most popular being stocks, the ways for people to invest in these markets, and the increased complex amount of regulation around trading. In the 1980s and 90s, the stock market was experiencing various booms, not just in value but also in a renaissance of technological evolutions. For example, in the 1980s and 90s, the trading environment was that of a physical floor rampant with loud yelling, chaotic movement, and constant manual-trading of orders. A manual trade would essentially be a stockbroker getting a request to buy or sell shares. The process of the stock broker trading would usually be done in the following:

1.) Verbally Outcry and Negotiate Bids and Offers Regarding Trades

2.) Negotiate through a Complex System of Hand Signals indicating the trade's intentions and thus executing quickly

3.) Recording Trades onto Paper Tickets, then take these tickets to a telecommunication terminal, processing the trade, and then clearing it

This was a process that could easily be misinterpreted, mishandled, or, rather, be improvised tremendously. Around the late 1980s, the rise of limited computers and phones allowed traders to be a lot more efficient but more importantly, allowed more trades to be done, thus bringing down the overall cost of trading (NYSE, 2014).

In the 1990s, the actual evolution of trading and market technology came into play with the rise of Bloomberg terminals, online trading via the internet, and the commissions pricing being lowered to incentivise online trading via up and coming online brokerages such as "AmeriTrade" and "ETrade". This was the beginning of Algorithmic Trading and high-frequency trading as stock trade was now allowed in increments of the dollar, down to 1/16th of the US dollar. More and more people were beginning to invest online, and with the rise of powerful personal computers, more and more people were looking to see how they could maximize profitable investments while minimizing risk. Eventually, the NYSE went fully public in 2005, and in the years after, many brokerages came onto the market where people could start making accounts and trade from their home (NYSE, 2014).

1.2 Stocks

A stock is a financial instrument that consists of shares that represent the partial ownership of a company. Stocks are represented with a ticker symbol which represents its own unique identifiable string of characters within its respective market. So, Nvidia's stock would

have the ticker symbol of “NVDA”. Each stock will have various values depending on different time frame intervals under the following:

1.) Open: the price at which the stock opened at for that specific time frame/interval
2.) High: the highest price the stock’s price traded at for that specific time frame/interval
3.) Low: the lowest price the stock’s price traded at for that specific time frame/interval
4.) Close: the last price the stock traded at at the end of that specific time frame/interval

Each stock has subsets of these values for each specific timeframe, so for example, there are time frames representing yearly, monthly, daily, down all the way to increments of seconds and even raw tick data which represents one specific trade. This can get easily complicated as this is a lot of data that can accumulate very quickly.

Regarding investments, one can typically buy or sell a stock. Even though there exists a multitude of complex and dimensional ways to invest in stocks, buying and selling a stock is the most common method. Investments occur from a portfolio which is a collection of the positions in stocks one has and the various history and data for each position.

It is better to look at investing in stocks as entering and exiting a position. When entering or exiting a position, the investor needs to make sure they can come out with a profit. However, that is often easier said than done as stock markets exhibit volatile and almost random behavior. This has led to investors relying upon special types of orders for the sake of entering and exiting an investment in case the price moves to a wanted or unwanted level.

Even though “long” and “short” are two different types of positions, they call upon similar order types for the sake of entering and exiting them, respectively (Hayes, 2024a). Orders in the stock market are sent when the user is ready to enter/exit a position and require a requisite number of shares for that specific stock. For the position status of entering or leaving to be finalized, the order must be filled. However, sometimes orders can be partially filled due to there not being enough shares in the pool of buyable shares known as liquidity. However, in more retail investor cases with algorithmic models, this should not be a problem as trading institutions with enormous amounts of capital deal with this.

Regarding order types, there are different versions and combinations, but the two main order types are “market” orders and “limit” orders (Reiff, 2025). “Market” orders do not have a set price, only a set number of shares. Basically, the moment any order for an opposite position is offered, the market order will match that order and whatever price it dictated. So for example, if an investor places a market order buying 10 shares for stock “X”, and somewhere at the exchange, the broker finds an order for selling 10 shares for stock “X” at a specific price such as 51$, the investor will then buy 10 shares of stock “X” at 51$. Market orders build upon the randomness and variations within investing in stocks. The same can be said for exiting a position as well.

The other type of order for entering and exiting positions is called a “limit” order. Basically, limit orders are specified by the number of shares like the market order, but unlike the market order, limit orders have a specific limit price (Reiff, 2025). This limit price dictates that the order can only be filled at a price level of the limit price or better. So for example, if the investor wants to go long on stock “X” for 10 shares, but is apprehensive, the investor can

initiate a limit order of $49.5 representing that the investor wants and only will purchase 10 shares of stock “X” if its current price is $49.5 or less.

These are two of the main order types and there are multiple variations of these order types offered in the framework to help facilitate accurate trading model design.

1.3 Building a Trading Model

With trading having evolved substantially, investors have looked for ways to maximize investments. This effort has led to the rise of algorithmic trading models, otherwise known as trading systems or trading algorithms. Trading models are computer programs with a set of logic for entering and exiting a position. Just like any investor, the trading programs seek to make profitable investments while minimizing risk and any money lost from these investments. However, just like with every technological evolution, the level of complexity within investing, the markets, and the tools also increases.

Building a trading model has evolved dramatically to the point where it is easier to create models that perform poorly and harder to build models that perform well. There are a myriad of steps and issues that one must deal with; however, one of the most basic rules to building a trading model is having a trading strategy.

A trading strategy is a set of rules for entering and exiting positions in various stocks (Hayes, 2024b). It is essentially a mathematical model that capitalizes on a trading pattern. A trading pattern is typically a specific, observable instance that repeats. Now, since trading models are about automation with a set of rules for entering and exiting positions, it is key that building a successful trading model requires a robust trading strategy supported by proper examples of consistent trading patterns in the stock market. Trading strategies also include different types of

entry and exit orders for initiating and exiting the position and risk management which is used to manage both profitable and non-profitable positions (Hayes, 2024).

There are numerous types of strategies, and not all strategies are equal. Some investors may simply want to go long and hold, meaning they buy stocks and keep them for a long time and then sell at a higher price for a bigger profit; or, some investors may want to incorporate different types of evidence-based analysis or data into their strategy. Since a trading strategy is a mathematical model of some pattern/phenomena in the market, it can take a lot of different forms. Some of the forms include the following but are not limited to or are a combination of: technical analysis, fundamental analysis, news-based trading, or day trading (Hayes, 2024a).

The overall goal of all trading strategies is a 2-step goal: predict the stock movement, and then capitalize on that prediction via a profitable entering and exiting of a position. Trading strategies need to be consistent, but the problem that arises is that stock market data is randomized. Even though there exists manipulation in the stock market, there are a lot of trades going on at any given second due to such large technological advancements that it can be overwhelming. A simple strategy with simple criteria, such as the close price crossing a 10-day moving average, can result in very disastrous outcomes because there is so much “noise” in the stock market. This is due to volatility. Volatility is defined as chaotic and rapid changes in the price movement of a stock. Volatile movement in the stock market can lead to a lot of false signals, otherwise known as noise.

1.4 Stock Market Volatility and Complexity of Parameters

Stock market volatility is where the complexity of algo-trading truly shines. It is not enough to have a strategy with observable patterns because there is so much rapid and chaotic

movement in the stock market with so many stocks. This is otherwise known as a market uncertainty and affects all financial trading assets (Bhowmik & Wang, 2020). It is not enough to simply know by seeing the patterns on each stock that those patterns and only those specific patterns can and will occur. Ultimately, algo-trading requires a greater application of trading knowledge, programming knowledge, and mathematical analysis to find and capitalize on proper trading patterns for a strategy to work.

One extremely crucial aspect of trading strategies is the parameters for the strategy to operate. A parameter in a trading strategy is a specific piece or variable that takes in a particular value that is used in the actual set of rules for the trading strategy. Strategies can evolve to be a very complex set of parameters but even the most basic of strategies require parameters. Selecting the right parameters for a trading strategy is another layer of making sure a trading strategy and trading model is profitable. Parameterization can be a complex task that can be computationally expensive as well and requires being handled properly by the investor to not waste time in finding the right parameters (Qin et al, 2021).

An investor may have a separate strategy where if the close price of stock X crosses above its “10-day moving average”, open a long position, and if the close price of stock X crosses below its “10 day moving average” and in a long position, then sell the shares and close the long position. In this example, the key parameter is the number of days for the moving average which is 10. When evaluating a trading strategy, there are a multitude of questions to ask, some of these questions are derived from parameterization - finding the best parameters that fit the trading strategy. In the case of the 10-day moving average strategy, how does the investor know if 10 days is the best number of days for the moving average calculation? Why not 11 days, 25 days, or 50 days? This fundamental question of best parameters that optimally fit a

trading strategy adds another dimension of complexity when evaluating a trading strategy and finding a trading strategy that works.

What if an investor has a robust trading strategy that works? Just because a trading strategy works now does not mean that it will work indefinitely. The problem with assuming that a trading strategy that works now will keep on working is because “ideas stop working”. Time-Series data representing the stock market is a “non-stationary process” where properties of that time-series data change overtime; the key idea being the aspects and extent of that data that change are unknown till it occurs. Stock market volatility is the reason that a strategy that once was consistently profitable can easily flip to being consistently unprofitable. This is why trading strategies need to leverage the technologies of today such that they can be robust and profitable despite encountering consistently unseen, random data.

Before one can build a trading model, certain aspects need to be present for the sake of the model being able to run/built. Some of these key aspects are the programming language, the hardware environment, and the brokerage account to trade. The programming language for building a trading model is key as this is where the investor will be able to take their trading strategy and automate it such that the program will process the stock market date and automate trading per the strategies’ rules. The hardware environment needs to be a powerful computer that can run fast and have a stable and fast internet connection. This is needed so that when all the new data for the stock market is being streamed, the model can take, save, analyze that data in quick and proper timing while the data is still accurate. Finally, the investor needs a brokerage account. A brokerage is where investors can go make an account to trade, store money, and observe live market quote data typically. This is what the trading model will use to make trades regarding entering, exiting, and maintaining positions for the investor.

Once the trading strategy has been decided upon and the trading model has been built, the problematic problem now arises: Is the trading model profitable? Yes, stock market data is a non-stationary process, extremely volatile, and overall noisy. However, since a trading strategy is a preset of rules for trading that capitalizes on a pattern, investors can utilize various testing methods to explore their trading model and strategy and understand how to make their model robust or if they need to create a new trading strategy.

1.5 Backtesting and Optimizing a Trading Model

One form of testing for trading models is called backtesting. Backtesting is taking past financial time-series data for a stock, applying the current trading model to that time-series, and observing the results through specific metrics. This testing method allows investors to see various insights on their model(s) performance, such as comparing multiple strategies, validating a specific trading strategy, or assessing risk. Ultimately, backtesting will enable investors to begin simulating their trading model to establish some form of empirical evidence regarding the model's performance.

Backtesting takes a set of parameters for the trading model, and will apply those rules to a set of time-series stock data in a specific range of dates where each increment is known as a bar. The application of the trading model to the series of bars, from the start date/time to the end date/time will see an application of trades that could have been possibly made had the model been live trading during the same period. Since this is historical data, the simulation of the model trading is not to be trusted entirely, as certain aspects of live trading cannot be replicated completely in most cases (Bailey et al., 2016). Backtesting will yield metrics that the investor can then use to analyze how their model performed; some popular metrics are the "net profit",

“average winning trade”, “sharpe ratio”, and many other ratios of varying complexities that look to analyze different facets of the trading model’s performance during the backtest. Generally speaking, a proper rule for evaluation is to backtest a strategy after its live performance and compare the activity of the trading model across both contexts as 1:1 as possible.

Since backtesting yields results by running a trading model with a specific set of parameters once, it is ultimately not enough to only use backtesting. There are multiple reasons for this. Backtesting is meant to be a simulation on previous financial-time series data from the past, it will never replicate the behavior of using the same trading model live, 100%. One of the biggest reasons for this is that for every time increment, whether it be a 5-minute bar or a daily bar, when it is live, each trade is building up that bar till the end of the time increment. For every bar of historical data, the high and low values represent the highest and lowest price for the stock during that specific increment; when live trading in the market, and observing the current bar that is building with trades, an investor can discern whether the high or the low of the bar came first as the bar is done building. However, when using past historical data, most trading models will not be able to see whether the high or the low came first for that bar unless the data source provided that explicitly. This is because the data is limited and only provides flat values at the time of that bar. This is one of the biggest examples of why backtesting and further simulation of the trading models need to be done in a specific manner with respect to constraints such as this.

Another aspect of building and evaluating trading models is optimization which is related to the parameters of the trading strategy. Besides the question of whether a trading pattern in a strategy is robust and consistent enough, a trading model needs to have parameters that are optimal and best fit its core trading strategy. By having the optimal parameters for a model per the available training data, performance for a model can be improved substantially (Xue et al.,

2021). For example, using a strategy for going long with the 10-day moving average, the investor needs to answer whether the strategy should use 10 days for the number of bars back, or any other number. This is where optimization comes in. If a backtest is a trading model simulation with a specific set of parameters and unique results, then optimization is a series of backtests/runs where each run will use a different set of parameter values specified by the user.

An optimization on a 10-day moving average strategy would be where the investor would specify values for the bars back for the moving average from any specific range, so for example, 4 to 20. For every integer value between 4 and 20, the user would run the moving average strategy, but it would use that specific run's integer value instead of 10. This would result in 17 different backtests, each with a different parameter value for the number of bars back from the moving average and different results to then analyze the backtest. This is where optimization of a trading model's parameters can shine because sometimes all it takes is tweaking the parameters to find out if a trading model is profitable or potentially its profitable parameters.

1.6 Overfitting in Trading Model Optimizations

With backtests and optimizations being tools for developing a trading model and evaluating its efficiency, there exists certain problems that cannot be mitigated with these tools alone: overfitting.

Overfitting is a very serious and common problem in building robust, profitable trading models; oftentimes it is the major reason why most trading models do not work well and lose money. This is due to the fact that the stock market is a non-stationary process and therefore having constantly fluctuating patterns of data. A trading strategy is what fuels the trading model, and its goal is to capture a proper trading pattern, consistently. This also means not capturing the

wrong trading patterns in stocks, no matter how similar the pattern may be. It is very common to encounter false signals with trading models which leads to poor performance in the stock market. Overfitting can be defined in many ways but it is when the model has been fitted to the training dataset such that it gives falsely good results because of only adapting to the trading dataset.

In the case of trading models, overfitting results in false signals for entry or exiting positions which is when the stock will exhibit movement not entirely but similar enough to the trading strategy's rules/logic. These false signals signal a potential position or exit; when in reality, the false signal is not a true logical signal per the strategy; it is just noise in the market. Having situations where false signals are generated despite a model being backtested and optimized "well" is where a trading model and trading strategy's true instability and weakness is shown.

Finding the root cause for these false signals and for overfitting can be difficult due to the myriad of causes. Some of the biggest problems with backtest and optimization is that both operations use the same dataset for evaluation and testing of the trading strategy. When running both of these operations on the same dataset, a trading a model that runs well on that dataset may seem to do well via result metrics, however when it comes to live data which is random, the model may end up doing poorly. For example, if the investor's backtests and optimizes their original trading strategy of the 10-day moving average and finds per the results that a 4-day moving average is better,they may choose to employ that strategy with that specific parameter of 4 bars back. Unfortunately, when the trading model goes live with that version of the strategy, the stock price could easily fluctuate where the close price goes above the 4 day moving average and then drops significantly and the model could potentially lose a lot of money. Where the model would have predicted a profitable long signal based on the 4 day moving average, the stock

market can easily shoot down after entering the long position via that signal and show a significant loss. This can lead to disastrous outcomes for the investor as the model, despite showing good results on the previous historical data, performs very poorly when live.

1.7 WFA (Walk Forward Analysis) a Solution to Overfitting

While overfitting is a severe issue with trading model development, and has many different causes, there exists a solution that can help remedy overfitting, walk-forward-analysis (WFA) optimization. One big issue with backtests and optimization is that the same dataset is being used for all the testing and evaluating. The problem here is that the model is not being tested on any unseen-data, or rather, out of sample data. All the optimization and backtests are done on in-sample data that is then evaluated. When trading live, the trading model will be encountering new, unseen data every bar/time increment. This data can be seen as data that is out of sample. This is important to acknowledge because a trading strategy that is robust in nature will be able to decipher the new,unseen live market data into proper signals or noise that will not trigger a false signal and thus enter an improper position (Unger, n.d.). WFA is the solution to this problem of trading models and the data that models are trained and evaluated on.

WFA combines two of the methods of evaluating strategies, backtesting and optimization. Firstly, the entire historical dataset is separated into multiple sections where there is an initial segment referred to as the in-sample data window (Unger, n.d.). After the initial segment, the remaining part of the entire historical data set is separated into further mini segments called out-of-sample data windows (Unger, n.d.). For the initial in-sample data window, the trading model is optimized on that segment of historical data only, not any of the other data windows (Unger, n.d.). This is so the trading model can be trained on that initial data

section; however, there is an objective function that must be specified to sort the optimization runs for the best parameters. For example, if the trading model's objective function was set to "Net Profit" then after the optimization runs on the in-sample dataset, the parameters belonging to the best run per the highest net profit are recorded and used. From there, once the optimization is done and the best parameters are ready, the model will then run a backtest on the next window, the out-of-sample data window. The out-of-sample data window represents the unseen data that is used to evaluate the model. Just like the live market data, the training model will not know about this data ahead of time, so it is up to the robustness of the trading strategy within the trading model to then act accordingly to the unseen data from the out-of-sample window. After this backtest is done, the results, parameters, and start and end dates for the backtest are then recorded. However, this only represents one round of WFA optimization. When the dataset was broken up into an initial in-sample data window, the other part was broken up into a series of out-of-sample windows. So, after each round of WFA, the out-of-sample data window for that specific round is added to the in-sample data window, and the next out-of-sample data window will be the next data window for the backtest (Unger, n.d.). This is known as anchored WFA and it works by 2 core aspects: increasing the in-sample data window size by encompassing the prior out-of-sample data window every round so there is more data to optimize parameters; and, by moving to the next out-of-sample data window, the model will not only be optimized on a larger dataset, it will be back tested on newer more unseen data (Unger, n.d.).

Each new round of WFA will be using more training data in its in-sample period, and testing new data which helps to reduce overfitting significantly. Each out of sample data window is different from the other windows, and by incorporating shifting to new Out-of-sample data windows each round, differing market conditions and dynamics for the model to trade in is

simulated. For example, say an investor has a strategy where they go long on a series of stocks in their portfolio. The ideal conditions for that strategy would be once in positions the price of each stock goes up. However, the market is a non-stationary process and because of that, a crash or correction can occur and cause the market to drop, leading to all the prices for the stocks in that portfolio to drop significantly. It is easy for an investor to say how their long-only trading model works in conditions that suit long-only trades, but what about situations like a crash or correction which are the antithesis to long-only trading strategies. Would the long-only trading strategy still buy on the way down or would it stop buying if its logic is robust enough? If the investor never trained and tested their model properly using WFA on the market conditions where conditions for the model to underperform occurred, the investor would not be able to understand if the trading model is able to perform well in varying market conditions that are ideal and not ideal for its strategy. A model getting the right trades is not enough as the model needs to avoid bad trades as well and manage risk properly when dealing with market data live. Since both backtesting and optimization on only the same dataset are static and do not show the differing cycles of market conditions, WFA is able to circumvent this such that it can lead to trading strategies that are credible and consistent in the constantly-changing atmosphere of the stock market.

1.8 Significance of Thesis

The goal of this thesis is to provide a tool for investors to build robust trading models. Overfitting a trading model always leads to disastrous outcomes as stock market data is a non-stationary process. By providing a tool that allows investors to run WFA on their trading models, investors will be able to see how their trading model and trading strategy can be improved in response to out-of-sample data. Overall, this tool will allow investors to extrapolate

meaningful insights on their trading model and pursue data-driven choices for their trading model.

1.9 Thesis Structure

The structure of this thesis is organized into the following sections.

Chapter 1: Introduction will introduce the concepts of stocks, algorithmic trading models and overfitting trading models. This section will focus on the problem of overfitting in the trading model and the significance of tools like this framework in solving said problem.

Chapter 2: Literature Review will review existing literature in algorithmic trading. Different types of bias in the trading model development process, selection of specific trading indicators, and the process of backtesting,optimization and WFA with regards to trading models will be focused on.

Chapter 3: Framework Introduction provides an overview on the framework, its capabilities, and its requirements. This section is meant to primarily introduce the framework's goals.

Chapter 4: Framework Design consists of the methodology and design philosophies behind the core aspects of the framework such as data access, configuration files usage and formatting, and other aspects integral to the capabilities of the framework

Chapter 5: Framework Implementation describes the software tools, programming techniques, and choices for implementing key features of the framework. This section will constitute implementing the majority of the design aspects discussed in Chapter 3.

Chapter 6: Framework Demonstration and Analysis presents and discusses a demonstration of using the framework to explore a trading model's performance. This section

will show and explain examples of configuration files and a trading model being optimized and back tested in the framework. Furthermore, this section will use all the result data generated by the framework to visualize the performance of the model and extrapolate key insights to bolstering the trading model tested.

Chapter 7: Conclusion summarizes the framework and provides insight on the challenges faced during development, wish list for features.

Chapter 2

Literature Review

There has been a substantial rise in algorithmic trading models development due to an increase in availability of stock market trading, services, and technology. This section will highlight some of the reasons that inspired this framework and provide accurate information on the current landscape of algorithmic trading.

2.1 Rise of Algorithmic Trading

With stock trading becoming more computerized, there has been an increase in demand for models that trade stocks properly. One big problem however is the rise of volatility in the market with algorithmic trading which can make it harder to make robust models. As referenced by (Damilare Oyeniyi et al, 2024), algorithmic trading models moving in the market have the ability to increase liquidity 10% due to how fast these models can trade. Furthermore, because of the fast speeds of trading models, trading markets in response have become extremely volatile depending on significant world events or markets. This results in powerful algorithmic trading models being able to move markets by placing large orders for stocks in such a small amount of time and any trading algorithmics that are not built properly to adjust for this volatility can get swallowed up in the trades and lose significant profits if the wrong trading signals are generated. An algorithmic trading model is supposed to be stable and when volatility is induced in the market by other algorithmic trading models, this can lead to false signals and capital loss due to noise.

Another problem attributed to the rise of algorithmic trading is that strategies that are generally popular do not always work well. It is interesting because why would the strategy be popular then? Simply put, it is because too many models/investors are using said strategies. This does not mean common strategies are bad, however, it takes a unique strategy not entirely being employed by many investors for a trading algorithm to make consistent profits. A popular strategy is the RSI (Relative Strength Index) which is meant to identify reversal points in stocks price data based on a series of ranges, mainly an upper and lower bound, of 70 and 30, respectively. Whenever a stock is below the lower bound of 30, the RSI dictates this as timing for an upwards movement of the stock price action. A lot of investors that use the RSI in their strategy will use it as an indicator of momentum and turning points in a stock's price action. However, according to (Hill, 2019), "RSI range alone is inadequate because it does not always capture upside momentum. The RSI range measures trend consistency well, but a momentum component is needed to uncover the strongest uptrends.". It is important to realize just like with the RSI, many components of strategies have different parameters and different uses that can be altered slightly in value but provide varyingly different results. In this case, by providing a separate momentum indicator, the RSI can work well as it is better for trend strength/consistency instead of trend identification via momentum which is the more popular use case. Just because the RSI is a popular strategy does not mean it is used properly and by making sure to vet popular trading indicators first can they then be used in a strategy.

2.2 Biases in Trading

Many other causes can lead to overfitting, such as bias in the development process of the trading model. Bias in a trading model can come in many different forms, but ultimately results in the same issue: overfitting of the trading model.

One core example of trading model bias is “data-snooping bias”. The trading model development process is extremely data-driven, as any changes that cannot be supported by data should not be made. However, sometimes the data used or generated by trading models can be manipulated in a way that skews the results and creates a positive illusion of performance. For example, when constructing portfolios of stocks, specific test statistics are used to evaluate the portfolio's performance with respect to a stock’s market valuation (Lo et al., 2015). Market valuation can be defined as the price at which the stock is sold. The problem here is that stocks with a higher market value may not perform as well in the number of trades won over the number of trades lost compared to those with smaller market valuations - where trades won is trades made by the model with a profit..They can however generate a higher amount of profit due to their higher market value. This tends to lead certain stocks that are not as strong performers to perform well or vice-versa because of a model’s fitting to the noise of the data. By focusing mostly on market valuation, performance of the trading model can be skewed due to poor stock selection. Certain false assumptions can occur, leading to falsely statistically significant findings on the model’s performance for certain stocks when it is just noise that the model is adapting to due to poor stock selection and analysis. It is essential to select the appropriate metrics when analyzing a portfolio of stocks that a trading model will trade.

Sample selection bias in trading model development can manifest in many different forms. One key example is selecting the stocks to trade in the portfolio. It is essential to analyze

stocks in terms of various characteristics to ensure they are suitable for the portfolio; however, some characteristics are more commonly known than others. One example of this is that the cost of sending orders for specific stocks can vary compared to orders for a different stock in a portfolio. Capital constraints are a significant part of the algorithmic trading model, and trading costs can accumulate quickly, especially in models that execute a high volume of trades. It was actually found that using characteristics such as market capitalization, share price, order size, and order type for stocks in the NYSE can lead to higher net costs for even facilitating a trade compared to other markets (Bessembinder, 2003). This does not mean dropping or adding stocks due to the expense of trading specific stocks in certain markets, such as the NYSE; however, it provides another component: examining the order selection and order routing of stocks for a trading model across different markets. The same stock can be traded for significantly less in terms of trading costs and with better order fills at two different markets. However, if this is not known, model performance can be skewed by a buildup of trading order costs and poor fills due to slippage. It was further found that the limit price executed for limit orders at the Chicago Stock Exchange was considerably lower for the same stock at the NYSE (Bessembinder, 2003). Having differing order fills and order costs can negatively skew the analysis of stocks performing in a portfolio for a trading model if not acknowledged and understood.

Evaluation bias is a type of bias where the results are greatly influenced depending on how they are generated and viewed. In the case of trading model development, there is a key metric that is always used to evaluate trading models performance. According to Arakelian, et al (2024) “. Using inappropriate performance metrics to evaluate the model also causes bias”. This essentially means the right metric(s) must be used in evaluating the performance of a trading model, otherwise an investor may make the wrong inference on the ability for a trading model to

perform. For example, if a trading model is ran in a singular backtest, and the only metric looked at is the "Net Profit" of the model, the investor may think the higher the net profit the better the model; however, this is a false assumption that can lead to disastrous outcomes for the trading model. If the model has 100 trades, where 99 trades were a loss, and one extremely profitable trade that eclipses the loss from the 99 trades, the investor may falsely assume this model is accurate and ready to be traded live. A separate model could have 100 trades where 99 trades were profitable, and there was a sizable loss on one trade which could have dragged net profit down substantially. Does this mean the trading model is a poor performer? The answer is impossible to know without further testing of the model. This is why it is important that models be evaluated properly and tested against the correct trading metrics so that bias in evaluating the model can be decreased substantially and allow proper, logical extrapolations of the model's performance to occur.

2.3 WFA in Literature

Walk-Forward-Analysis consists of optimizations on an in-sample period and testing/validating results from the in-sample period via the out-of-sample period. WFA can be a substantial tool in combating overfitting of trading models and this is due to a variety of reasons. In the stock market, there can be many different market conditions that can influence the overall behavior of the market. This distinct behavior of the market is called a market regime and during a market's life cycle, multiple market regimes can take place. An example of two market regimes would be a substantially upwards movement and a substantial crash/downwards movement in the market. Having trading models being prepared to combat these different market regimes that can appear is crucial and through walk-forward-analysis, a trading model's performance and

adaptability can be validated in response to these differing market regimes (Chojnacki et al, 2023). Having a trading model become adaptable to changing market conditions and have similar performance or at the very least not severe performance drops or profit losses is an integral sign of a model's robustness.

The whole point of walk-forward-analysis is to have an empirical method of validation for trading models. The stock market has too many nuanced aspects that will never be modeled, let alone by individual investors. It is important to use the existing tools that an investor has to engineer a trading model. While WFA can absolutely help in over-fitting, it serves its purpose as a tool for validation of trading model results above everything else. As stated by Arian et al (2024), the goal of incorporating validations tools for trading models is to "bridge the gap between theoretical robustness and practical reliability in financial models, enhancing their applicability in high-stakes financial decision-making, from asset allocation to risk management.". By having models that are validated properly, the trading model process becomes logical and validated by data generated from the tool.

Chapter 3

Framework Background

With the problem of overfitting being so common in trading models, this framework was developed to provide WFA (Walk-Forward Analysis) as a tool to anyone looking to create robust trading models. This section will serve as a guide to the framework's fundamentals, capabilities, and requirements.

3.1 Framework Objectives

This framework is meant to allow users interested in algorithmic trading to provide their own algorithmic trading model to then backtest, optimize, and conduct WFA. A central part of software in the finance sector is the programming language the models and software are usually run on since speed and efficiency are of the utmost priority. This framework builds upon that notion and is built in C++ for the sake of speed since C++ is a lower level language and thus will perform faster than models and backtesting libraries that are built in other languages that are higher-level and more readily used such as Python or Javascript - where programs are executed at a much lesser speed generally. Furthermore, the goal of the framework is to allow investors to simulate their model under a variety of different conditions and have the proper tools to conduct the analysis needed to generate proper insights about their model and thus trading strategy. Most importantly, this framework is meant to be a tool of simplicity for anyone interested in

algorithmic trading model development as there are many different libraries that can offer a lot more tools than are needed.

3.2 Framework Capabilities

The capabilities of the framework are as follows:

1. User-Provided Trading Models
2. User-Provided Parameters for Adjusting Trading Model Simulation
3. User-Provided Model-Backtest Simulation
4. Order, Position, and Risk Management Market Aspects Simulation
5. Report Generation of Trading Model Run Results
6. Multi-threaded Optimization for Large-Scale Trading Model Parameterization
7. Anchored Walk-Forward-Analysis Optimization for Trading Models

3.3 Framework Requirements

There are three requirements for the framework the user must fulfill to use this framework:

1. User-Provided trading model in C++
2. Configuration file(s) of user-specified model and simulation parameters
3. Financial stock data

Each of these requirements have major uses throughout the framework and are discussed in Chapter 4: Framework Design and Chapter 5: Framework Implementation more in depth regarding their uses, formatting requirements, and rationale. Ultimately, this framework provides

optimal tools for users in their efforts to build robust trading models that can succeed in the stock market.

Chapter 4

Framework Design

The process of developing the framework required a lot of modeling first, then followed by programming to make sure many different aspects were created properly. This section will detail not all, but the majority of the core design aspects of the framework. This includes the design rationale behind how the framework was developed, certain design choices and tradeoffs made, and the various aspects of trading markets modeling required for trading model simulations in this framework to function properly. Some code is referenced and used to explain certain design choices, but overall, most of the implementation and programming is explained in Chapter 5: Framework Implementation.

4.1 Framework Architecture

This framework is based on the EDA (Event-Driven Architecture) design pattern. It allows real-time processing of events, communication between different states asynchronously, and scalability such that modifications can be done properly. The reason this style of the framework was adopted was due to how trading models behave when live trading. For example,

a trading model when live must be able to source and process user-provided parameters, ingest past and current financial data for calculations, and manage different positions in stocks all while being operated from a central container. This is how trading models operate on a series of brokerages where investors can deploy their models, but also on proprietary and/or cloud-based setups. A trading model can be considered a complete system of pieces, each accomplishing different tasks and yielding results, all of which may be dependent on the other pieces of the system. Because of this complex nature of events, EDA was chosen as the central design pattern for the framework.

4.2 Framework Skeleton

The skeleton of the framework can be separated into separate pillars that represent the specific parts of the system:

1. Simulation Engine → Central Container and Coordinator of All Event-Driven Processes
2. Configuration File Manager → User-Provided Configuration Files and Parameters Management and Accessibility
3. User-Model-Interface → User-Provided Model for Trading Simulations
4. Data Handler → Central Data Handler for User-Provided Data Reading, Storage, and Access
5. Activity Logger → Activity and Error Logger
6. Trading Simulation → Event-driven simulation of Market Dynamics such as Portfolio Management, Orders and Position Generation and Execution, and Report Generation
7. Optimization and WFA → Handler and Container for Backtest Results, Parameterized Optimization Runs, and WFA Results

These pillars represent the overall areas that the framework is built on. Each pillar is meant to be the container of its respective activity that will then work with the other pillars to ensure the user can have seamless and robust development of their model.

4.3 Simulation Engine

Of the core pillars, the Simulation Engine represents the overall container for all the events of the framework. Whereas simple backtests can be run without any optimization in the framework, nothing can be done in the framework without the simulation engine. The Simulation engine acts as the key foundation for all the activity since it houses class data members that represent all the pillars. The Simulation engine is meant to coordinate all the activity between all the other pillars and allow proper access to everything.

The simulation engine is where the following activities will occur:

1. Loading, Access, and Maintenance of Configuration File/s and Parameters
2. Loading and Access of Stock Data
3. Application of Trading Model Calculations
4. Preparation and Execution of Trading Model Simulation, Optimization, and WFA
5. Portfolio Tracking and Managing of Positions
6. Trading Signal Generation
7. Order Generation and Execution
8. Position Tracking and Execution
9. Report Generation

The simulation engine's goal is to provide proper functionality to all of these tasks as the sole coordinator and ensure the events and interactions all occur in the proper order to maintain the

proper state of the trading model simulations. As the lynchpin for activity, the simulation engine will conduct everything from loading in a user's configuration files and trading model; to reading in and storing the stock data in memory; to advancing simulations and then generating reports for analysis.

4.4 Configuration File Manager

The configuration file manager pillar is meant to be the pillar that represents the user's configuration file/s in memory and allows access and maintenance of all parameters. The goal of the configuration file manager is to provide trailering of the market simulation per the user's specification. To accomplish this goal, one of the files the user is asked to provide at runtime is the configuration file. The user can also provide multiple configuration files referenced in one configuration file.

4.4.1 User-Provided Configuration Files

The user-provided configuration file is a TOML file with specifications regarding default parameters required by the framework and separate parameters completely custom and designated by the user.

4.4.2 TOML File Type

TOML is a file type extension that stands for "Tom's Obvious, Minimal Language" and ends with an extension such as ".toml". The reason the TOML filetype was chosen is because it is easily parsable and readable and allows the storing of parameters and settings but also the simplicity of customization as well. TOML Files take the form of <Key> = <Parameter> and

allow many different native types for the user to specify such as: Arrays, Integers, Floats, Booleans, and many other types. This allows the user to specify everything from simple parameters of one value to groups of parameters such as:

1. SIMULATION_USER_NAME=”YUSSOF KAZMI”
2. INDICATOR_PARAMS=[1, 2, 3.14]

Ultimately, the usership of the TOML File Type provides a lot of simplicity when it comes to storing parsable data in a consistent format.

4.4.3 Configuration Files Usage

Configuration files are used in software predominantly because they separate the settings from the application logic. For example, hard-coding values that can change frequently when needed such as number of bars back for a calculation or unique API Keys for requests is considered very poor practice of software design, but for good reason. Everytime a program needs to run, it needs to be compiled, and everytime any amount of changes are made, the program will have to be recompiled. Having values that are hard-coded that need to be changed frequently in a program that is running means that that specific value needs to be re-edited everytime and thus the program will then have to be recompiled. This can lead to large amounts of unnecessary time spent waiting especially for larger programs. Furthermore, having changeable, viewable values that are very sensitive values but hard-coded is not good practice for security reasons. For example, API Keys are generally stored in a separate part from application logic because if users can see that value whether in the code or behaviors within the program, that value can then be taken and used without permission or for other nefarious reasons. This is why configuration files are needed as they add a layer of abstraction for

user-specified parameters such that it is useful on many different levels for program execution and security.

4.4.4 Default Parameters and User-Specified Parameters

For the framework to run properly, the user needs to provide at least one correctly formatted and correctly populated TOML configuration file. While the formatting is universal to the TOML file-type standards, the user needs to obey a rule of the framework regarding the parameters and that is the Default Parameters for the simulation. Part of the framework's goal is properly simulate market dynamics when a trading model needs to be backtested and this resulted in the following list of default parameters that must be obeyed:

1. HIST_DATA_DIRECTORY
2. HIST_DATE_REF_TICKER
3. QUANTITY
4. STARTING_CAPITAL
5. TRAILING_STOPS
6. PROFIT_TARGET
7. START_DATE
8. END_DATE
9. REPORT_DATA_PATH
10. REPORT_BACKTEST
11. REPORT_DAILY
12. REPORT_WEEKLY
13. REPORT_MONTHLY
14. REPORT_ANNUAL

15. REPORT_TRADESLIST

16. LOG_FILE_DIRECTORY

With these default parameters, the framework can ensure it has what it needs to then run properly.

Another part of the logic for the configuration files the user provides is the special parameters beyond what the framework itself needs. For example, the user may have special parameters for their model, such as variables needed for calculations. The user can then elect to either define these parameters directly in the same configuration file as the default parameters or create a different configuration file with all the parameters and values separate. If the user elects to do the latter, they only need to put the file path of the second configuration file in the first configuration file. When the framework reads in the initial configuration file and encounters the file path of the second configuration file, it will automatically open that specific configuration file and recursively read and store the parameters from that file as well. This allows the user to have different parameter files that are abstracted away with the values that align with that file. For example, during the development of this framework, there were two configuration files: main_configuration.toml, which had the default parameters, and parameters.toml, which had the model-specific parameters for calculations.

4.4.5 Access of Parameters

Once the user has loaded the configuration files, and the parameters are all processed, the simulation engine will expose methods to allow access to those parameters. Throughout any point in the simulation, the user can then invoke methods to grab those parameters and properly use them; whether for calculations, error checking, or conditional logic, the ability to do so

becomes completely up to the user. This ensures the user can properly and accurately customize their simulation to be as realistic as their trading model set up when running live in the market.

The important part about parameters is that they must be used properly and sometimes there are instances of parameters either not being used at all, or used at wrong times. To remedy this, whenever a parameter is accessed, it is then timestamped such that the user can display the parameters at the end of simulation and see if specific parameters were accessed properly.

4.5 User-Model-Interface

One interesting challenge during development was designing and developing the proper way for a user to provide their model to the framework. The simulation engine was meant to be the container of everything and so having the model come into that container as an interlocking piece was an interesting system design problem to solve. The approach taken was to develop an Example Model class as a derived child class of the simulator, and from there the user can then provide their own model based on certain stipulations.

4.5.1 Extending the Simulation Engine Design

Since the simulation engine contains all the processes and access capabilities for the simulation, providing a model that is a direct extension of that design will allow the user-provided model to conduct processes seamlessly. The logic for this design follows:

Simulation Engine → Example Model API → User-Provided Model

This decision accomplished two major goals with for the framework:

1. Abstraction of Logic
2. Simplification of Model Development

Regarding the first goal, there is a lot of infrastructure and logic in the Simulation Engine that, while used by the user indirectly, does not need to be edited and changed. For example, there are methods that focus solely on the core event loop for simulations, or methods that focus on dynamic memory management for objects not even referenced directly by the user; having the user be able to accidentally edit such things can lead to disastrous outcomes that would require tedious bug fixing.

4.5.2 Example Model API

Regarding the second goal, the user will need to provide their own model. Now, since the Example API Model is used, it provides a template of a series of virtual functions that the user can then replicate in their model. There are eight main virtual functions that the user will need to implement, per the Example Model Class:

1. Apply_Calculations → Applied predefined calculations to all stock data
2. Apply_Model → calculates and generates signals for entering positions
3. Generate_Long_Exit_Signal → calculates and generates exit signals for long positions
4. Generate_Short_Exit_Signal → calculates and generates exit signals for short positions
5. Create_User_Entry_Order → customization for order types of entering positions
6. Create_User_Exit_Order → customization for order types of exiting positions
7. Generate_Trailing_Stops_Amount → customization for risk management via a trailing stop
8. Generate_Profit_Target_Amount → customization for risk management via a profit target

At first glance, this may look barebones, but realistically it is truly only what is needed when the user wants to provide their own model. For example, a trading strategy's anatomy is the following:

1. Logic behind the entering and exiting of positions
2. Various variables parameters and calculations
3. Order entry and exit customization
4. Risk management customization

This is where having the user-provided configuration files is beneficial the most. If the parameters used in the logic are recordable, then when the user provides their configuration file full of all of the default and custom parameters, that takes already a lot of work away when providing the model.

In the eight functions listed above, the ApplyModel method will be where the user will generate their calculations and checks for entry conditions, where as the Generate_Long_Exit_Signal and Generate_Short_Exit_Signal will be where the user will generate calculations and dictate how and when they want to exit any existing long or short position. This allows the abstraction of all the other aspects of setting capital, setting the commission, and many other tedious aspects that do not need to be explicitly done by the user in code. Furthermore, since the user-provided model is a child object of the simulation engine, then that means the user-provided model will have access to all the public methods that the user may need when it comes to accessing parameters and other data members when needed.

4.6 Data Handler

Data is the source of every trading model's decisions as the trading pattern within a trading strategy is always based on trends within the data; this leads to handling,accessing, and storing the data being of extreme importance. However, before any data can be processed and

analyzed, the user must provide one last piece to the frameworks besides the configuration file/s and trading model: the data they plan to conduct their model's simulation upon

4.6.1 User-Provided Data

One of the default parameters required is "HIST_DATA_DIRECTORY" which is a file directory path to a series of csv files that the user provides the simulation. Each csv file for the stock must follow the following rules:

1. Columns must be the following:
    a. DATE
    b. ADJUSTED_OPEN
    c. ADJUSTED_HIGH
    d. ADJUSTED_LOW
    e. ADJUSTED_CLOSE
2. Date Column must follow the format: "MM/DD/YYYY", no intraday data is allowed
    a. This is so the date is properly parsed and processed into the framework

Having these stipulations for the stock data allows the framework to read in and parse the csv files at the data directory specified and store all the stock data properly for each ticker. Stock data is usually specified as OHLC which stands for Open, High, Low, Close. Most data brokers will have data in csv format of this standard however there are instances where prices can be different due to a variety of reasons. Some examples are mistakes in the data depending on the data provider, prices being raw and unadjusted, or events such as stock splits and dividends. Having adjusted stock data is key to circumventing a lot of these differences and aspects. If adjusted data is not used then historical performance will be inaccurate due to events, and there cannot be consistent comparison of a stock's current and past price values. A key part of

algorithmic trading model development is making sure the data is sourced, cleaned/formated, and saved properly and by having adjusted data, these issues are solved.

4.6.2 Data Handler Architecture

The Data Handler pillar of the framework can be divided into three different segments, each responsible for their own tasks, but representative of the common goals for the pillar: Allow proper parsing, efficient storage, and non-redundant access of financial data the trading model simulations depend on.

4.6.3 Instrument Data Blocks

The user-provided financial stock data needs to be stored in a proper, accessible way. The design chosen for this task was to emulate something called a " Instrument Data Block" for each stock. Since each CSV file is mandated to have the format of a data frame, the Instrument Data blocks are structured to sort data in such a format, but have extensible capabilities. For example, when dealing with times-series data, being able to index by the time axis in that data is key. By building custom instrument data blocks through object-oriented programming, specific methods are able to be created to then allow such actions and more.

Having custom data blocks that obey the template of logic makes them reusable for each CSV representing a specific stock since it is a central container for all that stock. The logic of allowing storage and access of financial data, specific columns and rows, and specific attributes allows the user to then interchangeably manipulate and run simulations on these data blocks in a robust, accessible manner. Furthermore, when events in the framework occur, being able to have access to instrument data blocks where the data is already in memory means that the data does

not need to be redundantly read in again and stored for specific operations; this allows proper resource management.

4.6.4 Instrument Data Reader

When the user has provided a valid configuration file, and loaded in all their parameters, one of the default parameters “HIST_DATA_DIRECTORY” will be used to source all the csv files from the instrument data reader. The Instrument data reader’s sole goal is to take the file directory path for “HIST_DATA_DIRECTORY” and read and process each of the csv files located there. For each file that is processed, it is turned into an Instrument Data Block representing that specific stock’s data file by its ticker symbol. This process allows the construction of Instrument Data Blocks and once each Data Block is structured, the Data Block is handed off to be inserted into the central point of data access throughout the simulator engine, the Instrument Data Access.

4.6.5 Instrument Data Access

Just as the simulation engine is the lynchpin to the entire framework, the Instrument Data Access is the lynchpin to all of the data accessing the framework and user-provided model needs. Any access to the data blocks that contains the specific stocks data must be done through the Instrument Data Access. For example, if the user were to need a specific array of values that represents a column, they would have to find the correct Instrument Data Block, and the correct index for that array of values. This can become a tedious operation and so the choice to design this was to develop a separate container of Instrument Data Blocks that would take care of all the accessing and manipulation of data blocks directly, the instrument data access. Whenever the

framework is run, all the user needs to do to reference any form of data is simply utilize the Instrument Data Access with the ticker and name of the column of values they are referencing.

Another important aspect of the Instrument Data Access is the ability to keep all of the data blocks stored once in the same spot. Manipulating all the stock data in memory without having to create a copy or read in more data is key for performance. A detrimental bottleneck that can occur in trading systems is that oftentimes a copy of the data will be made and then accessed instead of the actual data. For every simulation and method having to make and then use a copy of the real data means having more space taken up which combined with poor memory management can compound into poor performance.

To mitigate this, the Instrument Data Access only ever provided direct access in the form of references or direct pointers to the data blocks such that any change on any data block is forever and no copy of a data block is ever created for the sake of a task. Keeping the instrument data blocks access direct and concise is what allows the different pieces in the simulator to manipulate the data blocks whether that be inserting new calculations, or referencing specific details.

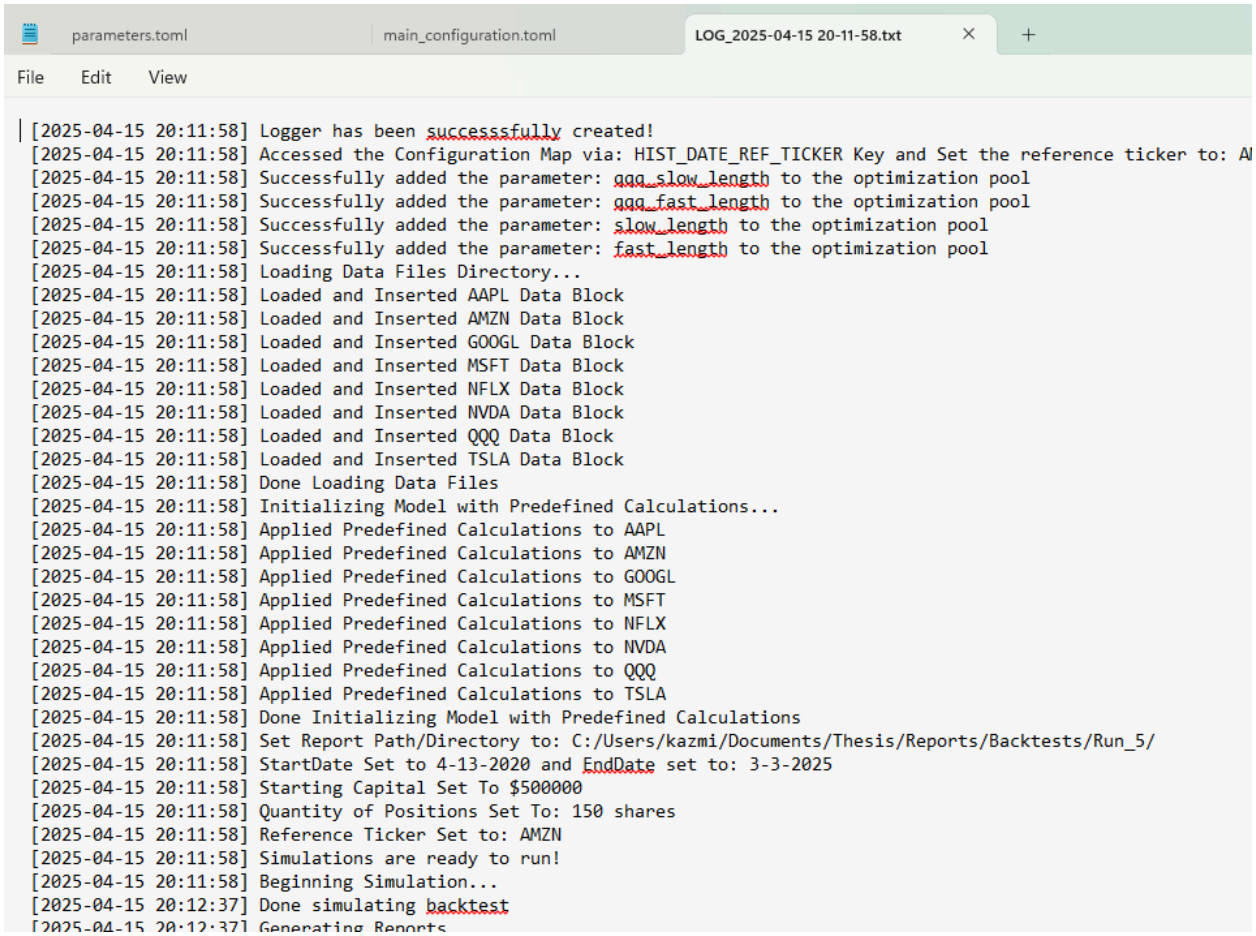
4.7 Activity Logger

Debugging programs and looking at past activity have been a key staple in software development. It is one thing for a program to crash and have an idea of what occurred at the last moment vs. having a collection of activity details before software that is running goes wrong, especially unexpectedly.

An activity logger in the simulator was a decision that was made later in development but realistically, should have been done earlier in development at the beginning. Having a logger can help accomplish the following:

1. Performance Monitoring
2. Debugging/tracking bugs
3. Understanding Features interacting
4. Recording of Past History

Proper activity logging via a logger can provide an accurate pinpoint and can minimize debugging time dramatically. For example, when it comes to the performance of a trading model simulation, it is important to understand how much time it may take to load in all the data blocks, or parse the configuration file for model parameters, or see how long a trading model simulation took to be run. Also, whenever building out a new feature or debugging a program it is absolutely crucial to have all of the details and analytics for how a new feature or current program is performing. Activity Logs can show what features are being used and how said features are being used. One key aspect of this during the framework development was the ability to see different trading signals during trading simulation detected and monitoring the capital and other metrics of a portfolio during simulation on a day by day basis. Another major aspect of the activity logs in software development is the ability to flush the current log and save it to a new file.



(Figure 1: TOML Configuration File Example)

This is crucial as date and time stamped log files of previous runs and iterations can help debug a program swiftly and properly as the previous logs of activity give an empirical baseline of evidence per what occurred and what did not during that specific run's lifecycle.

4.8 Trading Simulation

The core of trading model development is analyzing results from the simulations that are run with that trading model. Having the proper simulation for trading model's built is absolutely integral towards building a model's entry and exit signals, finding the right parameters without overfitting, and creating proper benchmarks of data for performance analysis. However, building

the logic for trading simulations and its constraints and trying to make it as accurate as possible is a complex situation that requires further modeling of many different market dynamics.

4.8.1 Key Market Simulation Dynamics

Part of building the framework requires building the trading simulation's which are integral to the backtests, optimizations, and WFA runs. If the trading model is the car, then the trading simulations are the race track, and like every race track there needs to be different aspects created and modeled properly such as roads. The following are some key market dynamics that must be absolutely present for a trading simulation to be logical:

1. Portfolio Tracking and Management
2. Trading Signal Generation
3. Order Generation, Queuing, and Execution
4. Position Generation, Closing, and Tracking
5. Risk Management

There are many other aspects of the market that can be modeled for simulations such as Bid-ask spreads, order books, and others; however, for the sake of properly simulating a model, these are the key aspects that must be developed with logical rationale and implemented efficiently.

4.8.2 Trading Simulation Event Loop

Whenever a trading model is simulated, there is a common logical event loop that always occurs.

Starting on the start date for the trading simulation, each day is passed through till the end date. During this range, each time interval of the dataset is broken up; so for example, in a dataset with 100 days, the start date could be day 1 and the end date can be day 100. The model

will iterate over each day then in the dataset, and ingest the new price information for the day. From there, the model will perform calculations with that specific day's data for each stock and generate entry signals if not in a trading position, or exit signals if in a trading position per the calculations for that specific stock. This is meant to simulate the model receiving new data for that stock and thus use it performa up-to-date calculations and generate signals. Once a trading signal is generated for entering a position for a specific stock, the signal will enter a signal queue. From the signal queue while still on the same day, the signal will be turned into a generated order to then be processed the next valid trading day. When the next day is advanced, the order will be analyzed by the portfolio. If it is an exit order, the portfolio will close the qualifying stock position by ticker and if the order is an entry order, then the portfolio will open a position under that stock's name.

4.8.3 Portfolio Tracking and Management

The portfolio is representative of the trading model's performance when being simulated or trading live. That is why the portfolio is one of the most important aspects that needs to be represented properly in a trading simulation.A portfolio can have multiple jobs, but in the framework it has a series of important jobs:

1. Start/End Date for Conducting Simulation
2. Tradeable List of Tickers
3. Container for Stock Positions
4. Tracking/Updating/Recording of Metrics such as PNL and Capital

With the portfolio having these jobs, it is important that the portfolio be built properly to enact said jobs. For every trading simulation run, there is only one portfolio present as the

portfolio for a given trading model simulation will be representative of that simulation's activity and thus be representative of the results for that simulation.

The first key aspect for the portfolio is having the specific dates for the simulation to be run during. Since the portfolio is the container for the results and trading model's activity, it needs to be in specific windows per specified by the user in the configuration file (ie. Start and End Date).

Another key aspect for the portfolio is the list of tradeable tickers it is responsible for. The portfolio must know each and every stock ticker it is trading so that the portfolio can keep track of all the proper stocks and their respective activity when being simulated in the market.

Furthermore, the portfolio will be the container for all the stock positions that open during the trading simulation, and also store the past positions for each of the stocks to keep lists of historical positions. Having all the stock positions represented in a central place like the portfolio will then allow the job of tracking and updating metrics for the various stock positions to be updated seamlessly and properly as the trading model simulations advance day by day.

4.8.4 Trading Signal Generation

When a trading model is simulated or live-trading, it will ingest new data, perform calculations, and based on those new calculations, generate entry and exit signals per its criteria. While the aspect of robust entry and exit signal generation is for the investor and their model to figure out; the handling of the signal and what happens next is handled by the framework.

Whenever a new trading signal for a stock is generated, there are a series of steps that can be taken depending on the type of signal. When an entry signal is generated, that means the model has calculated a signal indicating that a specific type of position should be entered based upon the newest data. The entry signal can be broken down into two types, a long entry signal,

and a short entry signal. Exit signals are the other type of signals broken down into two types as well, a long position exit or sell-short position exit signal. Entry and exit signals, no matter the type, are still handled the same in the sense that signals are taken in an order system where a specific order is then generated for that type of signal. The point of differentiation regarding signals is the order generation. For example, a buy entry order will be generated for a long entry signal, or a Buy-To-Cover order will be generated for a short exit signal.

4.8.5 Order Generation, Queuing, and Execution

One of the most important aspects of trading model performance is the order strategy for entering and exiting positions. Part of a robust trading strategy is having the right type of orders for entering positions Using the wrong order type to enter and exit positions can lead to massive losses. Emulating the proper order dynamics within the overall market dynamics is key to ensuring proper test runs. Building a proper order system into the framework required the modeling to be in depth and broken into three parts:

1. Order Generation → Taking a Trading Signal and Generating the Right Order
2. Order Queuing → Sending the order to be queued for the next time interval
3. Order Execution → Executing the order such that the proper position is opened

Part of the user provided model is also providing the right type of orders for entering and exiting orders via the User-Model-Interface. Allowing the user to provide the proper type of order logic they use in their model when it is live is key to making sure the model's trading activity is simulated properly.

When a signal is received, depending on the type of signal, a specific order is generated based on that signal.

Trading Signals will contain information for the order to be generated such as the following:

1. Stock Ticker
2. Quantity
3. Signal Type

Using this information, the order system can generate the proper order for entering and exiting positions.

Irrespective of the type of signal, these are the proper order types decided to be emulated:

1. Limit Order
2. Market Order
3. Stop Limit Order
4. Stop Market Order

Depending on the type of signal, certain orders can not be used or can be used. HOwever, each of these order types were emulated due to how commonly incorporated they are when trading models are built and deployed.

Limit Orders are a type of order for entering positions where the model will set a specific price for that order to be filled at. If the order is filled, then that means the price it is filled at will be the limit price or a better price. Depending on the type of signal and position being entered or exited, the ‘better price’ for the order can be higher or lower than the limit price.

Market Orders are a type of order for entering positions where the model will not set a specific price for the order; rather, the market will try to fill the order at the next available price. For example, if a market order is generated from a prior day’s signal, then that means it can get

filled at the stock's open-price at the next trading day's open. The price can be purely random depending on when the market order is generated and sent to the order queue.

Stop Limit and Stop Market orders are part of a special type of orders called stop orders. Stop orders are orders that can only be generated if in a trading position and act as a form of risk management. When in a position, the goal is to make the maximum profit for that position and risk management via stop orders is a way to ensure some type of profit is gained. Stop orders act as a type of level for profit protection in a position. If the price for that stock hits the level of the stop order set, depending on the type of stop order, the stop order will trigger.

Stop Limit orders are a type of stop orders in which there are two parts. The stop limit price and the limit exit price. Like the limit order, the stop limit order utilizes specific prices as levels; however, the stop limit order when triggered by the stock price hitting the level of its stop limit will not exit the position or fill the order. Rather, once the stop limit level is hit by the stock price, a limit order of the same price level of the stop limit's limit exit price will be generated and if the stock price crosses that limit order, the position will be exited.

Stop Market orders are a type of stop order that is typically used for emergency exit situations. Basically, the stop market order will have a specific price like the stop limit order to be set at; however, when that specific price level is hit, a market order will be generated which is then filled as soon as possible at a random price set by the market.

When orders are generated, they are then sent to an order queue system where the orders will then be filled at the following next bar/interval or canceled if not filled. This is a process before the order is filled and the framework will handle it by taking and holding the order for the next day to allow it to possibly get filled.

When orders are generated and queued, they are then checked to see if they can be filled. Whereas market orders and stop market orders are filled by the open price of the current day's data, the framework will check limit and stop limit orders to allow proper fills based on the limit price levels specified. The framework does not have the ability to give a random fill after the limit price for the order has been touched but future development will include a way to help randomise this to an extent.

4.8.6 Position Generation and Closing

Opening positions to make profit is one of the key aspects of market simulations and modeling it for the framework to handle required very specific design choices. The first biggest design choice was first modeling the generation of a position when an order is filled. To accomplish this, since the entry-order that is being filled already has information such as the ticker, and number of shares, and type of order and type of signal, the position being entered would then be derived from that information. The same logic follows for the second aspect of position modeling which is the closing of position. Since exit orders are derived from exit signals, and thus require the same data, having positions be closed by consuming the information of an exit orders allows the framework to handle closing positions while storing data of the exit order properly.

The final aspect of position modeling is the position tracking itself. In a given trading position that is opened, there are metrics such as realized-p&l and unrealized-p&l. The framework focuses on tracking and storing metrics such as this that are position centric within the position itself. That way every open position depending on the stock's performance as the simulation advances is able to track metrics for positional analysis while the position is open, and also when the position is closed. Every day that is advanced through the simulation, once all the

closing of positions is done, the portfolio will then look at each closed position and calculate or update further metrics based upon the collection of closed positions for that day. For example, if three positions are closed on a specific day for the situation, the framework will total all the net profit from each of those closed positions and add it to the total net profit for the portfolio.

Allowing the positions to be modeled as their own units that updated on their own after each day of simulation and thus new stock information is key. Having this logic allows the portfolio to only worry about updating itself with regards to trading performance when the positions are closed. Furthermore, when positions are closed, the portfolio will store all past positions for every stock to allow proper data collection for analysis.

4.9 Report Generation

Building robust trading models requires very in-depth analysis of results regarding a model's trading activity. It is very easy to get overwhelmed by the abundance of metrics available when analyzing trading models. Furthermore, only having metrics that analyze a trading model's performance is sometimes not enough; for example, some may want to see a daily or weekly performance report; or a tradeslist by security. Looking at all the different possibilities of results generation, it was of the utmost importance for the framework to be able to accompany different types of reports and let the user decide in their configuration file which reports they want.

4.9.1 Different Types of Reports

The following are the report options the framework handles and provides:

1. Backtest Results Report

2. Tradeslist Report
3. Daily Portfolio Activity Report
4. Weekly Portfolio Activity Report
5. Monthly Portfolio Activity Report
6. Annual Portfolio Activity Report
7. Optimization Report
8. WFA Results Report

Each of these reports represents different aspects of the portfolio performance and allows the investor behind the trading model to really see their trading model's results for each simulation from different angles. The reports that are available depend on the type of simulation being ran.

4.9.2 Backtest Results Report

The backtest result report is meant to display the results of the trading model's performance overall per a series of metrics:

1. Total Trades
2. Net Profit
3. Gross Profit
4. Gross Loss
5. Number of Winning Trades
6. Number of Losing Trades
7. Max Winning Trade
8. Max Losing Trade
9. Average Winning Trade
10. Average Losing Trade

11. Win/Loss Ratio
12. Expectancy Score
13. TS Index

These metrics represent a trading model's performance as a whole, from the start date to the end date, utilizing every security in its tradeable tickers list. The logic for a backtest result is to allow the trading model to be evaluated as a whole unit, instead of its analysis being broken down into other parts like in other reports.

4.9.3 Tradeslist Report

The Tradeslist report is meant to show all the trades the trading model made while being simulated. It is able to represent all the good and bad trades with various metrics by stock ticker. Having this type of report available to the user allows an aggregate of data that the investor can analyze. For example, if there is a consistently underperforming stock in the portfolio while the other stocks in that portfolio are doing well, the investor can proceed with further testing and analysis to determine if the stock is worth keeping in the portfolio or if there is something wrong with their trading strategy, etc.

4.9.4 Daily, Weekly, Monthly, and Annual Portfolio Activity Report

Despite having the backtest results report and the tradeslist report, which both analyze the model in their own ways, that may not be enough. Having a series of reports for the overall view of the portfolio's performance with respect to different time-axis helps analyze how well a trading model can perform over-time. For example, a trading model that is simulated for 12 months, can over perform for 8 of those months, but during the other 4 months the model can under-perform. One question that can arise from this context is which 4 months did the trading

mode under-perform? Was it during the first 4 months of the data set, or the last? Was it during a stock market correction that went on for 4 months? Or was it 4 separate months separated by 2 over-performing months? The amount of questions that can arise when evaluating a trading model's performance is staggering. Therefore, allowing the user to see a full outside-view of their model's performance from a day-by-day, week-by-week, month-by-month, or year-by-year basis will allow the investor to investigate the present-anomalies in their trading model's activity, if any.

4.9.5 Optimization Report

Another form of reporting for trading model activity is the optimization report which is then when a trading model is being optimized. Optimization runs reports are centered around the user-specified fitness function, and so the reports consist of two major aspects:

1. Each run is sorted by its metric, so the run with the highest metric per the fitness function is at the top, and the lowest is at the bottom
2. Each run is recorded with its parameters and backtest results

Optimization runs can be run a very large number of times depending on the parameters and because of that, results for each run can pile up. To alleviate this mass of data on the numerous runs, the user's specified fitness function will be used to sort the runs in highest to lowest order so the more important runs are seen first. Also, each run is recorded with the parameter values for each specified parameter for evaluation and then the backtest results for that specific optimization run. Designing the optimization report with these two aspects in mind allows the user to see which optimization runs did the best, but also the specific parameters that constituted some of those best runs which is the overall goal of optimization.

4.9.6 WFA Results Report

WFA Results Report is the last type of report that the framework can generate and only can occur when the user is running a WFA on their trading model. Since the logic of WFA runs is to run optimization on in-sample data-window, get the best parameters from that run, and then backtest on the out-of-sample data window, the backtest results along with the parameters from the optimization, and the date ranges for the backtest are recorded. Since the goal of the WFA runs is to analyze a trading model's performance out of sample with the best parameters, having a report encompassing those three aspects is what's important when analyzing the results of the WFA being run on the trading model.

4.9.7 Optimization and WFA

When a trading model is only backtested, that will yield results of the trading model with respect to one simulation. However, in optimization and WFA, the model is simulated many times to get a more precise look at its performance.

When a trading model is optimized, per the user-specification of the specific parameters to optimize, the trading model is backtested a number of times to find the optimal parameters. The framework's optimization capability treats backtests as repeatable units with configurable parameters. Thereby allowing optimizations to run very without a lot of overhead when testing different values for parameters. Another important aspect of optimization is the specified fitness-function by the user in the configuration file. It can range from the following metrics that are used in a normal backtest results report:

1. Total Trades
2. Net Profit

3. Gross Profit
4. Gross Loss
5. Number of Winning Trades
6. Number of Losing Trades
7. Max Winning Trade
8. Max Losing Trade
9. Average Winning Trade
10. Average Losing Trade
11. Win / Loss Ratio
12. Expectancy Score
13. TS Index

When running optimization, the model is back tested a number of different times with different parameters. Each of the runs represent their own unique identity consisting of results and parameters different from other runs. Having the fitness function specified allows the framework to rank individual optimization runs to show potentially optimal runs with specific parameters and their values.

Optimization can be very taxing due to the fact that many different parameter values can lead to a large number of optimization runs having to be run. To mitigate this, the framework has built in multi-threading for optimizations that are also run asynchronously. This allows the optimizations to be done rather quickly then synchronously with one run at a time. Furthermore, having multi-threaded optimization means that the more powerful the user's hardware the more it can be leveraged to lower the time it takes for simulations and optimizations to run.

WFA builds upon both backtests and optimizations. Since WFA can be broken down into two sections, training on in-sample data windows, and testing on out-sample data windows, the framework combines both backtesting and optimization to achieve this. Each data window is representative of the overall historical stock datasets loaded into the instrument data blocks. Firstly, the framework requires the specification of four data points, the IS_Start Date, IS_End Date, OS_Start Date, and OS_End Date. These represent two time-windows, the in-sample data window(IS window), and the out-of-sample (OOS) window.

The framework will take a trading model and conduct multi-threaded optimizations on the model through the IS window data set and use the best parameters from those optimizations to then run a singular backtest on the OS window and yield results. Since WFA's goal is to evaluate how the model will run on 'unseen' data which is akin to the model trading live, providing the backtests on OS windows will help yield results on model's performance closer to that of live trading than simply backtesting on a whole dataset. Furthermore, having the model be trained on the IS window allows the user to find the better fitting parameters for their model's performance. However, WFA is ran in multiple rounds of testing and this requires the data windows to change accordingly, otherwise known as Anchored WFA.

Anchored WFA is the process of running WFA on a trading model, but for every round, the OS window is added to the IS window, and a new OS window of the same number of days is then tested. Every round of WFA consists of an Optimization on an IS window of a specific number of days, and backtests on an OS window of a specific number of days. At the end of each round, the framework will do two things to prepare for the next round of WFA: first, it will calculate the number of days in the OS window, and create a new OS window that is of the same number of days, but a new part of the dataset; secondly, the framework will take the current OS

window and append it to the current IS window so the optimization is ran on a longer data set next round. By creating a new OS window every new round of WFA, the trading model is tested on consistently newer data. This will repeatedly allow the model to be tested against unseen data for the entire WFA as opposed to testing on the same out of sample data. Secondly, by adding the old OS window to the current IS window, the model will be optimized on a bigger in-sample dataset providing more robust parameterization.

Chapter 5

Framework Implementation

When it came time to implement the framework properly, I had to make multiple decisions regarding the usage of the underlying programming language, its capabilities, and its associated libraries. A project like this requires detailed planning of many different aspects from the implementation perspective alone.

5.1 Technology Choices/Design Considerations

5.1.1 Using C++ for Implementation

I decided on C++ as the core programming language for implementing the framework. The decision for this was due to a variety of factors:

1. Program Execution Speed
2. Memory Management
3. STL
4. Many Different Libraries

C++ is a language used in a variety of different landscapes. Everything from large-scale software development in the form of systems to financial applications that require the utmost speed, C++ is used in many other contexts. One of the key reasons for the usage of C++ is its speed. C++ is a lower-level language than Python or JavaScript, and this quality allows for more direct communication with the computer's hardware. More direct communication with the hardware

requires more explicit programming in the form of explicit declaration of data types, manual memory control/allocation, and more complex syntax that does not benefit from certain aspects of other languages like Python. Python does not require explicitly declared variable data types, as it is inferred at runtime and has a garbage collector for memory allocation and deallocation. The tradeoff, however, is that because Python is less complex for the user, the speed is nowhere near as fast as that of C++. Yes, C++ requires a more complex learning curve, stricter adherence to language rules and logical programming, and more verbose syntax, but in response, C++ yields immensely faster performance than Python or JavaScript.

5.1.2 Memory Management

Memory management in C++ makes it a clear contender for applications in finance, where a lot of data is used at any given moment, and speed is of the utmost priority. Even if a programming language is fast, when dealing with massive amounts of data, the language can slow down relatively easily. However, there are cases where managing the memory of the data structures that hold these large amounts of data can yield substantial performance boosts.

Dynamic memory allocation and deallocation allow the framework to work with data whose size is unknown at run-time. For example, when the user loads financial data, the framework will create an InstrumentData object representing a data block of that stock's price data and other attributes. Some users will have CSV files for each stock that can be 5 to 10 years of data, while some CSV files will be 1 to 2 years of data; this is a large discrepancy in file size, and thus, memory must be managed depending on the user's specifications. By allocating and deallocating this memory into data structures with pointers, only as much memory as is needed at runtime will be used.

RAII is “Resource Acquisition Is Initialization,” a memory management technique in programming languages. It dictates that when memory is dynamically allocated to an object, it will be released once its lifespan has ended. RAII is a proper programming technique as it ensures the memory used by an object, like Instrument Data, will be released when the Instrument Data object is no longer needed. This is done through destructors, which are called when an object is destroyed and explicit deallocation instructions are required, such as deleting vectors or closing files.

Pass-by-reference is a way of referencing variables that are passed into functions. When passing variables into a function, pass-by-copy occurs where the variable is copied and recreated inside the function’s space. This can be very problematic for two reasons. The first reason is that variables of substantial data size and are passed by copy will lead to unnecessary reallocation of large amounts of data when the variable is copied and recreated for that function space. The second reason is that direct access and thus modification of the variable is no longer possible. A big part of the framework is its event-driven nature of responding to different events with payloads of data stored in specific variables. For example, if an orderticket object representing an order is sent to the order queue, the framework will update that same orderticket object if it is filled or canceled. By not having direct access, when that orderticket object’s state is updated, the framework will not know if it is filled or canceled because it was a copy created from pass-by-copy. Pass-by-reference allows direct access to variables directly passed into functions, where the state of the object/variable must be maintained for proper response due to events. Furthermore, by passing by reference, a lot of space and resources are consciously saved due to the framework making sure a copy is not created without good reason.

STL stands for “Standard Template Library” and provides many implemented tools for the framework to utilize. Data structures are containers that hold data, and a key aspect of manipulating data in any program is segmenting it to specific containers that provide particular functionality. For example, InstrumentData objects represent a specific stock’s price data. However, to store the columns of data with the price data, the InstrumentData objects use a vector of vectors, where each vector is a dynamically sized container. Each vector will increase its size and store data based on the data provided at runtime. Data structures like vectors and many others from the STL allow readable and usable code for linearizing operations that would take much longer to implement.

C++ also allows the usage of many other libraries that have abstracted complex logical operations. Beyond the various data structures and iterators from the STL, many different libraries are used in the framework. Some examples would be the Chrono library for time manipulation, or the Async library for asynchronous operations with threads. Many of the libraries will be discussed in a later section dedicated to central data structures and libraries. Using C++ for the framework’s underlying core programming language is the right decision as it allows for flexible approaches to complex problems, especially when dealing with situations based on event-driven paradigms.

5.1.3 Object-Oriented Programming Design Implementation

One significant aspect of programming is modeling the proper aspects before implementation. Object -Oriented Programming is a form of programming that allows creating and generating objects meant to model real-world objects. Each object can be designated a user-defined data type with a class name that provides functionality to represent behaviors in the

form of methods. Furthermore, the ability to use objects with user-defined functionality allows the abstraction of logic particular to the framework.

Using the OOP paradigm, many classes were developed with specific functions and jobs in mind. The essential classes that will be discussed are:

1. Simulator Class
2. Configuration File Class
3. Instrument Data Access Class
4. Instrument Data Reader Class
5. Instrument Data Class
6. Keep listing the classes Good to enhance the list if there is time.

Each of these classes represents a specific part of the framework that is needed. For example, when CSV files for the stock data need to be modeled, objects of the InstrumentData class will represent that data.

A concrete example of OOP in the framework is the DateTime class, which allows information abstraction and the encapsulation of specific methods. Manipulating date data in the framework is crucial since the underlying data that drives the framework is time-series data. The DateTime class is meant to be a model of specific dates in the stock's data files, but also allows specific functionality in the form of methods such as:

1. Getter Methods such as GetYear(), GetMonth(), etc
2. Validation Methods such as IsLeapYear(), IsValidDate, etc
3. Boolean Operator Overloaded Comparison Methods

The date in a stock's data file represents a specific day within that stock's price movement; however, having a DateTime object modeled after each date allows manipulation via these methods. For example, by having a DateTime object constructed from a date string format of "MM/DD/YYY", the framework can access the different parts of that date, check if the date is a leap year or valid trading day, or compare the date object with other date objects via operator overloading. Abstracting this logic into the DateTime class also means that any further methods built upon the DateTime class will stay within that class, making the code uniform and clean.

Since a big part of OOP is reusable code, specific design patterns have emerged for logical and readable code. Design patterns are meant to be reusable, modular solutions when designing/modeling software that needs to be implemented. Design patterns are techniques that have evolved with the programming industry and are generally used in systems programming because of their system design principles. Since C++ is very OOP-friendly and statically typed, design patterns are often used. Specific design patterns were used when dealing with particular design problems to ensure logical design within the framework.

The framework utilized an important design pattern called the 'Singleton" design pattern. Singleton patterns are known as creational design patterns where specific objects are created with a specific mechanism in mind. For example, in the framework, the InstrumentDataAccess class is meant to be a container for all the InstrumentData objects allocated with dynamic memory. Because of how large the Instrument DataAccess class is, it would not make sense for the class ever to be replicated through various behaviors. For example, if the class's instance were to be passed-by-value into a function, a copy of the InstrumentDataAccess class with all the data would be created, leading to more resources being taken up by the class. Thus, more time is required for operations to be executed due to inefficient management of resources. To ensure

only one instance is created, the Singleton design pattern is applied, which dictates to the framework that only one instance of the InstrumentDataAccess class is present and no other instance is allowed. This ensures the InstrumentDataAccess class is properly maintained with one instance and allows proper use of resources so the framework does not slow down when running simulations that require manipulating the InstrumentData objects inside the class, each representing a specific stock's CSV data file. Furthermore, having the singleton pattern present means there must always be a global access point for the InstrumentDataAccess class so that only one copy can be created and accessed properly.

Another type of Design pattern is known as Behavioral Design Patterns, which are meant to dictate how objects interact with each other. The framework used two patterns to help object interactions: the Template Method and Observer patterns.

The Template Method pattern is a design pattern that shows a blueprint for derived objects of a class to follow via specific functions. The framework requires a user to provide their model in the form of a derived object of the Example Model class. However, in that Example Model class, specific methods are listed for the user to implement. These methods are functions used by the framework to simulate a trading model. By having the user provide a derived object of the Example Model class that overrides the specific functions listed in this blueprint, the user's model can then be correctly simulated. The derived class provides a specific implementation per the user's trading strategy but allows their model to be plugged into the framework and then simulated properly.

The Observer design pattern is a pattern with a central object known as an observer that, upon receiving updates to its state, will propagate those state changes to a series of other objects known as dependents. An example of this is in the Portfolio class, which hosts a collection of

classes for managing a trading model's performance during a specific simulation. Some of those classes are objects from the Position class, representing a specific stock position during a simulation. As the simulation is advanced, day by day, the portfolio class will receive new price data for each stock and will update positions automatically based on this new price data. The updates could either mean closing the position, opening a new position, or only updating the profit and loss information for a specific position. Still, the portfolio object will receive specific state updates to then propagate and send to the objects that depend on it for those updates. This ensures that a one-to-many connection is established with the portfolio and its objects. When a proper state update occurs regarding the portfolio class, all the objects are updated accordingly.

Operator overloading is a programming technique similar to OOP and its core principles. For example, in operator overloading with C++, specific and special member functions are defined for classes to allow further manipulation of user-defined data types. For instance, in the DateTime class, specific DateTime objects must be able to interact with each other in the form of comparison operators such as "==, >, <, or << outstream". By facilitating these interactions, objects of classes with user-defined data types can be manipulated like primitive data types of the C++ programming language. For example, when evaluating the expression of "3 < 7", the result is simple; however, when two DateTime objects, such as "DateTime(1,1,1970) < DateTime(3,20,2025)", the compiler may not know how to evaluate this expression. Using comparison operators for DateTime objects was crucial for the DateTime manipulation of time-series data. So in the framework, the DateTime class and other classes overload these specific operators so that they can be evaluated with comparison logic.

This means the DateTime class' functionality is extended and provides a more robust manipulation of the DateTime objects, but the implementation logic for this functionality is also

abstracted away. Furthermore, when the program is compiled, expressions with comparison operators that involve either the primitive data types or the property operator-overload data types are evaluated correctly. The compiler will know which functions to call with specific arguments, thus leading to compile-time polymorphism and no ambiguity that could cause issues when evaluating said expressions.

With the user having to provide a trading model, and the template method pattern being implemented with user-defined trading models, virtual functions were utilized to allow runtime polymorphism. When the framework accepts a user's trading model, the user implements specific functions for their trading model to function correctly, such as “ApplyModel” or “Generate_Long_Entry” condition. In C++, there is a reserved word called virtual, which dictates that a method for a class is to use the implementation at runtime if there is one. For example, the framework's goal is to let the user’s trading model be used, which means their definitions of the model functions, not the Example Model class. By declaring the Example Model class’s function as virtual functions, the framework will purposely prioritize and use the user-defined model's functions defined in their respective cpp file and not the Example Model’s function despite the same name and parameters. This approach allows the framework to achieve proper runtime polymorphism where the appropriate object for the trading model simulation is dedicated and used by the framework, but ultimately implemented by the user.

5.1.4 Usage of Data Structures

Many different data structures already built in C++ were crucial in creating the framework and its interactions properly. The following data structures were used.

1. Map → Key-Value Container that allows O(1) lookup time

2. Vector →Dynamically-sized sequential container of data with random access through indexing
3. Queue → O(n) Container of data with FIFO (first in, first out) logic

These specific data structures each represent specific containers that allow specific manipulation of data stored inside them, whether it is primitive types or user-defined types.

For example, maps were used in configuration file parsing, where each configuration file is structured as key-value pairs. Using a map means the framework could allow the user to have O(1) time complexity when looking up specific parameters needed throughout the simulation.

Vectors were used throughout the framework as a container for many different types of objects due to their nature of sequential, ordered storage. When the InstrumentDataAccess object was instantiated and loaded with InstrumentData objects, each object was represented as a pointer to a dynamically allocated InstrumentData object. A vector was created for this specific data type to allow sequential storage of these dynamic pointers in one central spot for the Instrument DataAccess object to store, maintain, and manipulate.

Queues were also used throughout the framework as a way of representing orders. For example, orders are represented by a signal threshold, which is used to sort trading signals in terms of strength. Higher signal thresholds would represent stronger signals, thus indicating that the signal is more favorable for a fillable entry order. By having a queue where the FIFO order pattern was observed, the orders first pushed into the queue would get filled or thrown out if not fillable. This allows a specific order object to be preserved by the framework with respect to the strength of signal thresholds.

Other data structures were used throughout the framework; however, these three were absolutely integral to maintaining proper data storage, access, and manipulation throughout the framework.

5.1.5 Error handling/Debugging

Error handling and debugging are two fundamental concepts when creating any type of program. Since the framework is a collection of many pieces of code of different sizes and complexities, adhering to both of these notions is key in ensuring the framework can be extended without the consequence of building upon buggy, improper code.

To build robust error handling, (It might be better to say you are reporting and handling errors using exceptions.) two special reserved words known as “Try” and “Catch” were used throughout the framework to allow handling of specific errors. “Try” consists of a specific code block running, and if an error occurs, the program will throw an exception, which the catch block will catch and then run specific code as a response. Using try and catch blocks throughout the framework helps to trigger actions that can prevent the framework from stalling and also return valuable information about the error thrown that triggered the catch block to run.

Debugging the framework required extensive use of two tools: an activity log and the Visual Studio call stack trace. An activity log as a logger object allows many specific messages depicting the program’s functionality to be documented. These messages often indicate details of where and when the framework is running at the time of logging a specific message. By combining activity logging with the Visual Studio call stack trace window, which showcases a program’s call stack of function calls before it crashes, a proper context is established of what the program was doing before a specific crash occurs.

Combining these three tools built the framework in a proper environment with logical error handling and detailed debugging.

5.2 Core Classes

While there are many different classes that are used in the framework, from STL data structures to user-defined data types, there are specific over-arching classes of the framework that are integral for its functionality. A lot of the user-defined classes that are separate from these classes are meant to be utility or custom data structures, whereas these overarching classes are directly involved with the framework's software architecture.

5.2.1 Configuration File System

Implementing the ability for the framework to represent a user-specified configuration file meant and provide access to parameters meant the following aspects needed to be developed:

1. Loading in of User-Specified Configuration File
2. Parsing and Storage of User-Specified Parameters
3. Type-safe access of Parameters
4. Timestamp parameters that have been accessed during the simulation

The ConfigAccess class has a default constructor and splits its jobs into multiple methods.

5.2.2 Loading Configuration Files

To load the configuration file, the class method *bool Load_Configuration(std::string Configuration_FilePath);* was implemented; it will take a user-specified configuration file URL as a string, open it, and parse it for parameters to process. Here, the framework will then call

various functions to parse and trim strings in the form of a <Key>=<Value>. However, if any of the values are different file URLs from other configuration files, those filepaths will also be loaded in and recursively parsed for parameters.

5.2.3 Storing and Parsing Configuration Files

Regarding parsing and then storing parameters from configuration files, multiple methods and data structures were implemented to help with both of these tasks. When a configuration file is loaded via Load_Configuration, each line as a string is read and fed into an event loop specifically for parsing it and then deciding what to do:

1. Trim the Line of all whitespace
2. Split the line into a string config_key and config_value
3. If config_value is a configuration file path, recursively call load_configuration
4. If config_value is an array of values, call bool Parse_and_Store_Array(std::string& key, std::string value);
5. If config_value is a specific value, call bool Parse_and_Store_Value(std::string& key, std::string value);

After a line from the configuration file is trimmed, it is split into two parts, a key and a value. Depending on what that value is, a specific function is called to parse it properly. If the value pertaining to that key is a filepath to another configuration file, the Load_Configuration file is called recursively to then open that file and parse the values inside it; if the value is an array of values, the values are parsed by the Parse_and_Store_Array function which will parse the values of the array and store; if the value is simply one value, it will parse that value and store it.

Once the values have been parsed, they will need to be stored properly. To store parameters, the ConfigAccess makes use of the map data structure, where the premise is

key-value lookup. In the best case, the lookup time is of O(1) complexity. The structure for the map is that of a string key representing the parameter name, and the value is represented by a class template called std::variant. Std::variant is a class template that allows the storage of one value from multiple data types in a predefined set. By adopting this practice, the ConfigAccess class can keep a central map data structure that can hold various data types so long as that datatype is in the predefined set of specified types. This also allows proper type-safe storage of parameters.

5.2.4 Type-Safe Accessing Configuration Parameters

When accessing a map data structure, the best-case lookup time is O(1) (Big O is O(log n), in which case, the key value is known. The same premise applies to the ConfigAccess map data structure; however, for type-safe access, certain details are implemented. In C++, generic functions are functions that can work with different data types in their parameters and allow a broader solution in certain cases. Since the user parameters can be of different types and stored in a central map container, it is important that the access of parameters be approached from a generic programming standpoint. For example, the user of the framework can request a parameter that is a string datatype, or they can request a parameter that is an array of string data types. Fundamentally, those are two different data types being handled, and to mitigate any errors at runtime, the following function and logic were implemented:

```cpp
template<typename T>
bool Get_Param(std::string& key, T& value) const {
    auto it = m_parameters.find(key);

    if (it != m_parameters.end()) {

        try
        {
            value = std::get<T>(it->second);
            return true;
        }
        catch (const std::exception&)
        {
            return false;
        }
    }
    return false;
}
```

(Figure 2: Generic, Type-Safe Parameter Getter Code)

Here, the Get_Param function takes in two parameters: the string key for lookup and a reference to an empty variable for the parameter value the user is requesting. So, if the user was requesting a string parameter variable, the user would declare a variable of the data type string and then pass by reference that variable into this function. Here, the function will try to get and cast the parameter to the right data type value; if that fails, the function returns false. This function returns a bool value, but if it returns true, the correct parameter value can be stored inside the value variable passed by reference. Furthermore, when used throughout the framework, the framework is able to continue working with that value variable since the function returns true. Here is an example of the function being used to gather parameters:

```cpp
std::string key = "REPORT_DATA_PATH";
std::string report_directory;

if (!m_configuration->Get_Param(key, report_directory)) {
    m_logger << "Could Not Get REPORT_DATA_PATH...CHECK CONFIGURATION FILE";
    return;
}
```

(Figure 3: Example of User-Parameter Access)

The key “REPORT_DATA_PATH” is passed into the Get_Param function of the pointer to the ConfigAccess object, m_configuration. If the function returns false, that means the simulation ends, and the reason why is logged. If the function returns true, that means the string variable, “report_directory” was able to be found and updated with the right parameter value and casted with the right datatype. Having an approach like this ensures that type-safe access for parameters throughout the framework is always ensured, and errors due to invalid parameters are easily mitigated.

5.2.5 Time-Stamped Parameters

The ConfigAccess class also has methods for displaying time-stamped parameters anytime during simulation. It is important for the right parameters to be accessed during simulation, and by accessing any parameter, it will become time-stamped for the user to see. This solves a major problem of a parameter not being used for whatever reason during a simulation and can provide further investigation for the user. To implement this, a separate map data structure was used, where it will be referenced by a string key, but it will store a struct which holds three different values: a string version of the value, a bool variable isAccessed, and a std::time_t variable called time_stamp which represents the timestamp the value was accessed. From there, the user can then implement a print_configuration_file_contents(), which will show each parameter and its struct’s values.

5.3 Data Handling

The data handling part of the framework is split into three classes: InstrumentData, InstrumentDataReader, and InstrumentDataAccess.

5.3.1 Stock Data

Each stock's price data in the CSV files required by the framework can be represented similarly to a Python Pandas dataframe, more commonly known as a numpy array of numpy arrays, where each column in that pandas dataframe has a specific string name and an array of values for that column. Data reading, storage, and manipulation via Pandas dataframes in Python are very common data science practices due to their speed and efficiency. However, since C++ is a compiled language, it is much faster than pandas, since it is a Python library. InstrumentData is a class meant to accomplish similar jobs to the Pandas Dataframe, but it is also meant to have added functionality with specific methods per the needs of the framework.

InstrumentData, at its core, is a class centered around the vector data structure. Its most important data member is the *std::vector<std::vector<double>> m_InstrumentData*, a vector of vectors. Vectors are dynamically-sized arrays, and by having a vector of vectors, m_InstrumentData can hold massive amounts of stock data like in Pandas, but be much faster. Any column needed can be indexed with random access by the outer vector, which holds all the columns together. All the inner vectors can be accessed immediately and also dynamically sized as further data is added to them. Instrument Data also has a separate vector for trading dates per its stock data CSV file called *std::vector<DateTime> m_Dates*. This vector holds DateTime objects for each day in the stock's data file. As the Simulation for a model progresses, day by day, the current date for the simulation is matched with the corresponding DateTime object inside m_Dates. This returns an integer value that can be used to index any of the inner vectors of m_InstrumentData and then return the corresponding value for that day of simulation from that inner vector specifically.

5.3.2 Reading in Stock Data

A class called InstrumentDataReader was created to read the stock data CSV files. This class's only goal is to create and send InstrumentData objects to the InstrumentDataAccess class, which is the central point for all InstrumentData objects and overall data access in the framework. Since InstrumentDataReader is meant to be a bridge between the InstrumentDataAccess class and the InstrumentData blocks it creates, its constructor is the following:

```
InstrumentDataReader(InstrumentDataAccess* InstrumentDataAccess_ptr):
    m_instrumentDataAccess(InstrumentDataAccess_ptr) {};
```

(Figure 4: InstrumentDataReader Constructor)

The constructor will only take in one parameter, a pointer to a dynamically allocated InstrumentDataAccess class object instance. InstrumentDataReader will focus on creating and populating dynamically allocated InstrumentData objects and when ready, will offload the pointers to each of those objects to the InstrumentDataAccess class.

When reading in a directory of stock data CSV files, InstrumentDataReader uses a library to parse CSV files called RapidCSV. RapidCSV is a header-only CSV parser file in C++ and provides reading of CSV files and their values with respect to their interpreted data type.

```
//Create and initialize the rapid csv file stream object
rapidcsv::Document data_file(dataFilePath);

//Now, we first get the headers, and loop through them to then add them
std::vector<std::string> columnNames = data_file.GetColumnNames();
```

(Figure 5: Integrating RapidCSV)

```
std::vector<double> data_vector = data_file.GetColumn<double>(columnNames[i]);
```

(Figure 6: Example of Using RapidCSV)

For example, by creating an object of RapidCSV with a file path to the stock data CSV file as a parameter, the CSV data file is read into memory with its values automatically cast to the double datatype for specific columns when accessed. Having RapidCSV to parse the stock's price data CSV file with the ability to dump each column into vectors of a specified data type allows seamless transition to creating InstrumentData objects.

1. Dynamically allocate Instrument Data Block for a specific Ticker and create a pointer.
2. Process all columns and Insert Columns into the DataBlock via the pointer
3. Use the m_InstrumentDataAccess pointer to insert the pointer to the InstrumentData object in a map of datablocks

This event loop is the logic for creating InstrumentData blocks and then adding each one to the m_InstrumentDataAccess pointer so the framework's become populated with all the stock data it needs. Ultimately, InstrumentData Reader becomes an effective bridge between creating InstrumentData objects and then transferring ownership of those objects to the InstrumentDataAccess pointer.

5.3.3 Data Access

The framework's final part of the data-handling implementation is the InstrumentDataAccess class. This class is one of the most essential classes in the framework and had to be appropriately designed. One of the overarching goals of this class is to store all of the InstrumentData objects that each represent a stock's price data CSV file. Oftentimes, these files can be large, which in turn can require a lot of memory to be taken up by InstrumentData objects. Since speed is of the utmost priority in the framework, certain precautions are taken regarding the InstrumentDataAccess class:

1. Singleton Design Pattern
2. Dynamic Memory Allocation/Management

InstrumentDataAccess is a class that follows the Singleton Design pattern. Singleton objects are objects that follow a specific creational pattern based on best practices in software engineering. Singleton objects mean that only one instance of that object can ever be instantiated during a program's life cycle. To accomplish this, three steps are taken in the framework:

1. The constructor for the class is private
2. static data member of the same type in the class is declared: static InstrumentDataAccess* m_pDataAccess and is declared a nullptr;
3. A special instance function for the class is created to instantiate an instance only once

```
static InstrumentDataAccess* Instance(ConfigAccess* a_cf) {
    if (m_pDataAccess == nullptr) {
        m_pDataAccess = new InstrumentDataAccess(a_cf);
        //m_config = a_cf;
        //Intialize_available_field_ids_stack();
    }
    m_DataFieldsAccess = new DataFields();

    return m_pDataAccess;
}
```

(Figure 7: Instantiating InstrumentDataAccess)

By privatising the constructor to the class, instances cannot typically be created. However, to make the initial instance, the class will need static methods that allow the usage of a class's methods without a specific instance. From here, the static InstrumentDataAccess data member m_pDataAccess, which is a pointer to an InstrumentDataAccess class, is declared to be pointing

to a nullptr. Finally, by using the Instance function, it will check if the m_pDataAccess pointer is a nullptr. If it is, it will instantiate properly with a pointer to a configuration file, which will begin the InstrumentDataAccess class.

The main goal of the InstrumentDataAccess class is to store and provide a global access point to the InstrumentData objects, which represent all of the stock price data required for simulation. Any manipulation, insertion, or access of InstrumentData objects must be done by interfacing with InstrumentDataAccess. To accomplish this, all the pointers to InstrumentData objects are stored in a map data member known as * std::map<string, InstrumentData*> m_TickerToDataBlock. Every InstrumentData block can be referenced to by ticker name, should it ever need to be accessed.

By providing a global access point to the framework through a singular pointer and forcing only one instance per the Singleton design pattern, the InstrumentDataAccess class is able to prioritize speed immensely. If the InstrumentDataAccess class, while full of InstrumentData objects, were to have its instance copied, that would mean another instance full of all of the same and large amounts of financial stock data would be created, and the more copies, that would tremendously slow down the framework. By adhering to this design pattern, situations that jeopardize the speed will never happen.

When simulating a trading model, a series of DateTime objects in an ordered fashion is required to simulate a proper trading window. To reference this trading window, two aspects needed to be implemented:

1. Specific Ticker is stored for determining valid trading dates
2. A vector of DateTime objects that contains all the valid trading dates for simulation

By having a specific ticker to reference all the trading dates, and then constructing and storing a vector of all the valid trading dates, any simulation running throughout the framework can access the vector of DateTime objects to then build the proper trading window required for that simulation. If there are 12 months' worth of DateTime objects, but each simulation needs specific different months, then each simulation can access the overall window of valid trading dates inside InstrumentDataAccess by itself. This is done to construct the windows specific to a trading simulation without impacting other simulations.

A big part of the InstrumentDataAccess class is how it is accessed from many different parts of the framework, at various times during the framework's lifetime. Whether that is storing the pointers to InstrumentData blocks created synchronously, or allowing read-only access to InstrumentData objects during asynchronous optimization runs, the overall goal of InstrumentDataAccess is to be an efficient data handler for all of the framework's needs.

5.4 Portfolio System

For every trading model simulation, a portfolio is needed to track and update the many different aspects of performance for the trading model. The Portfolio class has multiple jobs that center around facilitating and tracking the trading model's performance during a given simulation:

1. Track and store all trading positions during a simulation
2. Record and Store Trading Performance During a Simulation
3. Generate Trading Reports After a Simulation

Every simulation that occurs, whether it be simple backtests or hundreds of asynchronous optimization runs, will have a portfolio object pertaining to that specific run. To provide the user

parameters, the portfolio object will be connected to the framework's central configAccess object that is provided via the Simulator class. Some of the required parameters for the portfolio object are:

1. "START_DATE" represents the Starting Date of the Trading Simulation
2. "END_DATE" represents the Ending Date of the Trading Simulation
3. "STARTING_CAPITAL" represents the Amount of Capital at the Start of the Trading Simulation. There is a maximum capital amount of 10 million USD.
4. "REPORT_DATA_PATH" represents the directory to write all the reports

These parameters are checked for and then validated at the start of every trading simulation.

Since the portfolio has to manage many different aspects of a trading simulation, it was developed based on the Observer design pattern. Observer is a behavioral design pattern that emulates a one-to-many relationship between different objects. The portfolio object for a trading simulation will hold multitudes of objects from different classes and data structures, such as:

1. Position → Represents a live position in the trading simulation
2. OrderTicket → Represents an order to open or close a position
3. OrdersProvider → Represents an order queue system
4. TradesList → Stores Trading History for each stock in the portfolio
5. Vectors, Maps, and Structs

Each of these objects or data structures have their state that can be updated for a variety of different reasons during a trading simulation; for example, as a trading simulation is advanced day-by-day, the portfolio object will interact with new stock price data for that and that new price data can influence all the other objects inside the portfolio object; if an orderticket object representing a stop-limit order for a stock is filled per the new price data than that means the

portfolio object received price data and communicated that to the specific order ticket object and updated it thereby changing its state. As the trading simulation advances day by day, the portfolio object will use the new price data to update the state of the TradesList object and record any new trades that occurred after a full day of simulation. The portfolio object must act as an arbiter for all the other intricate parts of the trading market that depend on being updated per new price data; whether that be creating and queueing a new OrderTicket object, opening or closing a position object, or updating structs that represent trading performance on a given timeframe.

Tracking and Storing Positions

Whenever a trading position is opened, it is through an object of the Position class and is manipulated by the Portfolio class. All position objects are stored and managed by the portfolio class to allow position tracking in one central container without any interference.

```
std::map <std::string, std::vector<Position>> m_pastPositions;

std::map <std::string, Position> m_currentPositions;
```

(Figure 8: Storing Current and Past Positions Objects in Portfolio Class)

“m_pastPositions” will record all the past positions for access in vectors that are accessed by a string version of the stock’s ticker symbol, and “m_currentPositions” will store current position objects directly accessed, but also by the ticker symbol. The reason for this is due to the fact that separating positions that are live vs positions that have been closed and thus are not live makes updating the trading performance for each of these stocks’ symbols and thus the portfolio more refined.

```
void Update_Positions(InstrumentDataAccess* data_access, int day);
```

(Figure 9: Tracking and Updating Positions Each Simulated Day)

When tracking and updating positions' performance, the portfolio object will call a specific function at the end of the trading day to update all positions. This function will use a pointer to the InstrumentDataAccess class "data_access" to access all the new data to update positions, and it will use an index to represent the specific dateTime object and day currently in the simulation.

```
double open = data_access->Get_Vector("ADJUSTED_OPEN", it->first)[day];
double high = data_access->Get_Vector("ADJUSTED_HIGH", it->first)[day];
double low = data_access->Get_Vector("ADJUSTED_LOW", it->first)[day];
double close = data_access->Get_Vector("ADJUSTED_CLOSE", it->first)[day];

position->Update_Position(current_date, open, high, low, close);
```

(Figure 10: Updating Positions using InstrumentDataAccess)

Using "data_access," all the up-to-date price data for that specific position object is read, and the portfolio object will then use a pointer pointing to the exact position to invoke its own update function to send that price data over. The position pointer will update its own state via the price data that the portfolio object sent to it.

5.4.1 Recording and Storing Trading Performance

Since the portfolio is the collection of positions and their separate performances, it is important for the portfolio to be able to aggregate all of that data and provide it into a proper collection of the portfolio's overall performance, as a unit.

Even though each live position is updated at the end of every trading day, the realized performance of a position constitutes a major part of the portfolio's performance. Realized metrics in trading focus on the results of trades, such as the profit/loss of the position at the time of its closing. To get realized metrics, the performance of a position must be updated when it is closed, indicating the position is not live and no further performance is guaranteed.

To record realized metrics, the following variables are declared and updated in the portfolio object whenever the portfolio object closes a position:

1. m_grossProfit → Represents Total Revenue from Trading Positions
2. m_grossLoss → Represents Total Loss from Trading Positions
3. m_maxWinningTrade → Represents the Max Profit from a Trade
4. m_winningTrades → Represents the total number of trades with profit >= 0
5. m_maxLosingTrade → Represents the maximum profit Loss from a trade
6. m_losingTrades → Represents the total number of trades with profit < 0
7. m_totalTrades → Represents the total number of trades

There are many other metrics for analyzing a trading model's performance during simulation, but these are the core metrics that the portfolio object is useful to derive all the other realized metrics. "Net Profit" is the total realized profit from a trading model's performance, but after losses have been deducted, so in this case it would be m_grossProfit - m_grossLoss. By centering on these specific metrics, the portfolio object is able to record raw trading performance day by day.

Besides recording raw trading performance that is updated daily, the history of a trading model's performance is also stored. By being able to access a portfolio's performance on a time axis, users of the framework can then view their trading model's performance to better

understand how it is performing throughout the simulation. To accomplish this, two types of performance with respect to a time axis are recorded

1. Daily Portfolio Trading Performance by Date
2. Day by Day Trades List of Portfolio's Positions

Since each valid trading day in the simulation is represented by a DateTime object, the portfolio object is able to store Portfolio trading performance with respect to a specific trading day. Each trading day, if any positions are closed, the portfolio will use those positions to generate its performance for the day and store it. Just as the main realized metrics variables for a portfolio's trading simulation are updated throughout the simulation, one problem is that only the final values of the realized metrics are presented at the end of the simulation in a backtest report.

```
struct Past_Portfolio_Results {
    DateTime m_Date;
    double m_netProfit;
    double m_grossProfit;
    double m_grossLoss;
    int m_numberTrades;
    int m_numberWinningTrades;
    int m_numberLosingTrades;

};
```

(Figure 11: Storing Each Trading Day's Portfolio Performance)

To provide a way of recording the values of the realized metrics with respect to a time axis, the portfolio class utilizes a struct to store those same realized metrics' current values at the time of recording, where the DateTime object will represent the specific day for those specific values. This is helpful for a variety of reasons; simply put, a trading model can have good months and bad months, or years or days. By only seeing the final values for the gross profit or net profit, an

investor may not know that their trading model performed poorly most of the simulation and had a few good, largely profitable trades that masked the trading model's poor performance over time. By having the day-by-day representation of the portfolio's realized trading metrics, the investor can see how their trading model performed over time, giving them a larger overview of the trading model's performance over a group of securities.

When looking at a trading model's performance, the trades the trading model makes can always provide more information on how the model performs. The trades of a trading model show the behavior the model is making per its trading strategy rules for entering, exiting, and maintaining a trading position. At the end of each simulation day, if there are any number of position objects closed, the portfolio object will aggregate those positions to a tradeslist object to record the trades for that day. The tradeslist class is like some of the other utility classes in this framework (maybe list some here), where it is meant to be an add-on to existing data structures with extended functionality. These classes are meant to be containers for information moving throughout the framework with added functionality on top of existing STL data structures like vectors or queues, so store and manipulate the information in question more easily. The tradeslist object is instantiated inside the portfolio object, and it only exists to contain information about past positions:

1. Trade #
2. Symbol
3. # of Shares
4. Date Entered
5. Entry Type
6. Entry Price

7. Date Exited
8. Exit Type
9. Exit Price
10. Total PNL
11. Profi
12. Capital Run-up

Every time the portfolio object closes a position object, the tradeslist object will consume a copy of the portfolio object to extract the metrics information and store it in “m_tradesList” which is a vector of map data structures. Each map object inside m_tradesList will represent an actual trade made by the model during a trading simulation.

5.4.2 Generating and Writing Trading Reports

Once a trading simulation is done, the performance analysis must be done through reports on how the trading model performed. It is not enough to see a high net profit value, and think that the trading model is ready to be deployed because a high net profit could lead to the possibility of a lot of bad trades and one good trade or it can mean multiple bad trades and many good trades; this is why it is important that specific reports from different aspects of the trading model’s performance are generated to ensure the reviewing of a trading model’s performance is analyzed with useful and not redundant information.

There are different types of trading simulations in the framework, depending on the simulation being run, whether it be backtests, optimizations, or WFA runs; however, the core reporting of the portfolio class is responsible for is trading models' performance during a backtest. The portfolio can generate the following reports:

1. Backtest Metrics Report

2. Trades/Transaction lists
3. Daily Portfolio Performance
4. Weekly Portfolio Performance
5. Monthly Portfolio Performance
6. Annual Portfolio performance

Each of these reports represents different angles of a trading model's performance during a trading simulation, and depending on the user's configuration parameters, certain reports can be omitted if not needed.

5.4.3 Backtest Metrics Report

The backtest metrics report is based on the realized metrics for a trading model's performance, overall. During simulation, specific sets of metrics are recorded and updated for the trading metrics, but after simulation, more metrics are derived for a more descriptive view of backtest performance. The resulting backtest metrics report:

1. Start Date
2. End Date
3. Initial Capital
4. Final Capital
5. Total Trades
6. Net Profit
7. Gross Profit
8. Gross Loss
9. Number of Winning Trades
10. Number of Losing Trades

11. Max Winning Trade
12. Max Losing Trade
13. Average Winning Trade
14. Average Losing Trade
15. Win/Loss Ratio
16. Expectancy Score
17. TS Index

The resulting backtest results report incorporates each of these metrics, which help provide an overview of the trading model's final performance at the end of the simulation.

5.4.4 The Daily, Weekly, Monthly, and Annual Reports

Looking at the trading performance over time, with respect to different timeframes, helps to see when the model performs well or poorly. Since the portfolio object stores the daily performance of the model in the form of the Past_Portfolio_Results struct for each day, the same approach is taken for the other timeframes, but after simulation.

```
struct Weekly_Portfolio_Results {
    DateTime m_weekStartDate;
    DateTime m_weekEndDate;
    double m_netProfit;
    int m_numTrades;
};


struct Monthly_Portfolio_Results {
    DateTime m_weekStartDate;
    DateTime m_weekEndDate;
    double m_netProfit;
    int m_numTrades;
};

struct Annual_Portfolio_Results {
    DateTime m_StartDate;
    DateTime m_EndDate;
    double m_netProfit;
    int m_numTrades;
};
```

(Figure 12: Portfolio Results Structs Example)

Once the trading simulation has ended, and the portfolio object begins updating final metrics, it will take the overall day-by-day results of the model and segment the performance depending on the timeframe into specific blocks.

```
Week, Net Profit, # of trades
03/04/2020-03/06/2020, 0.000000, 0
03/09/2020-03/13/2020, 0.000000, 0
03/16/2020-03/20/2020, -351.660000, 6
```

(Table 1: Weekly Report)

```
Month, Net Profit, # of trades
03/04/2020-03/31/2020, -1624.650000, 23
04/01/2020-04/30/2020, -658.850000, 43
05/01/2020-06/01/2020, -386.840000, 66
```

(Table 2: Monthly Report)

```
Year, Net Profit, # of trades
03/04/2020-01/04/2021, -8481.350000, 246
01/05/2021-01/03/2022, -19336.410000, 525
01/04/2022-01/03/2023, -16989.630000, 759
```

(Table 3: Annual Report)

By creating structs that represent a specific week's performance, or a specific month's performance, or a specific year, each with their own start and end dates for that period, the performance of these timeframes can all be calculated and generated by the day-by-day recording of the model's performance.

5.5 Simulation Engine

The Simulator class is meant to be the central pillar of the framework. It is where all the major interactions by the framework occur and where influential objects are stored. Having these jobs means the Simulator class must have many methods that call upon different data members, provide clear access to specific data members, and be built with proper design patterns and clean code.

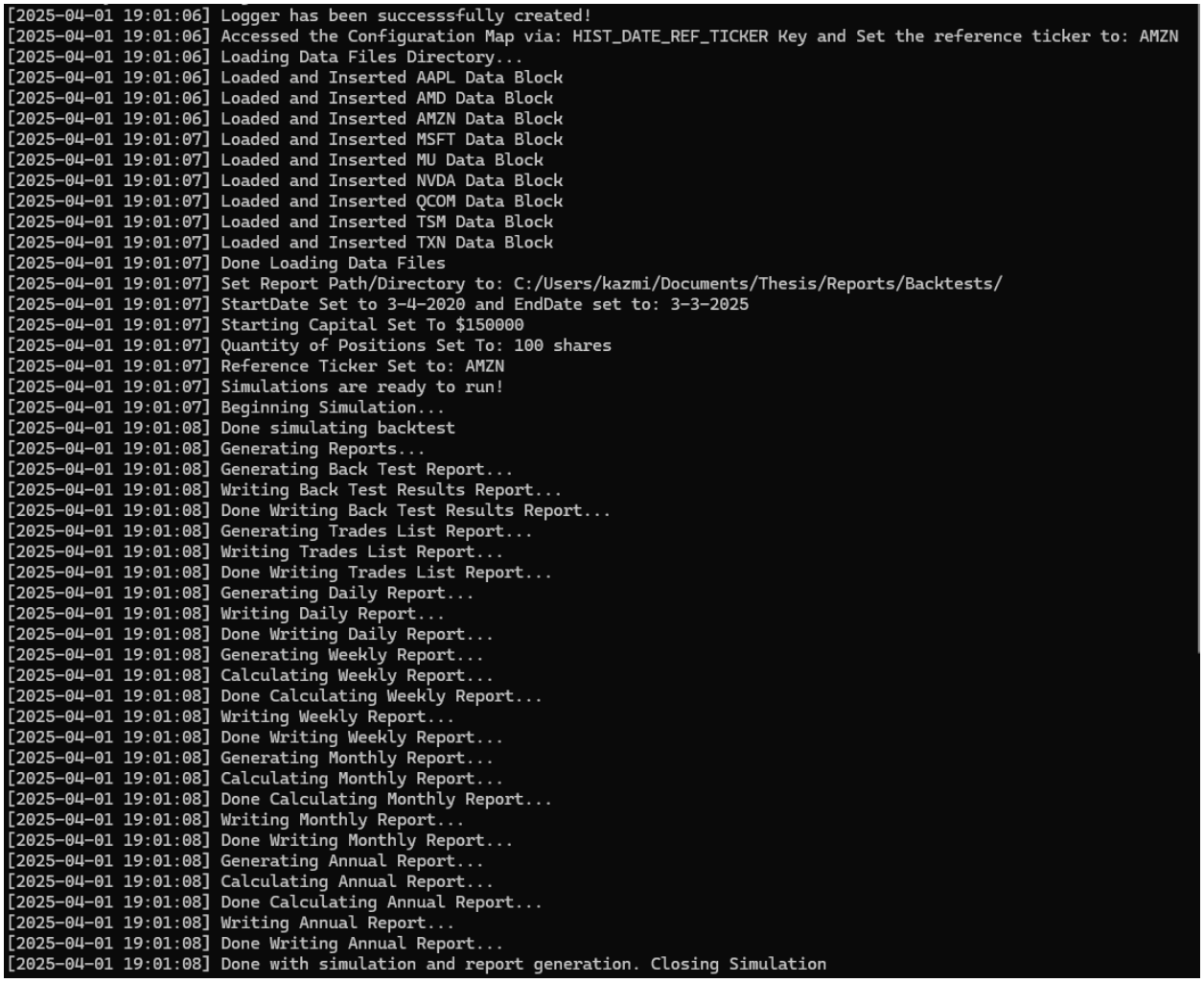
5.5.1 Specific Data Members for Specific Tasks

Since the Simulator class is the central focal point for the framework, some of its data members are very important to the overall functionality of the framework. The following:

1. InstrumentDataAccess* m_dataAccess
2. ConfigAccess* m_configuration
3. Logger m_logger

While not all of the private data members present in the class, these specific data members represent core aspects of the framework and are represented within the Simulator class. For example, since the InstrumentDataAccess class is constructed with the Singleton design pattern, where there can be only one instance, it makes sense to have the InstrumentDataAccess object, m_dataAccess, be constructed and stored within the Simulator class. However, since singleton objects require a global access point for manipulation, the m_dataAccess data member is a pointer to a dynamically allocated InstrumentDataAccess object. This allows the proper global access point throughout the framework for data access, storage, and manipulation. The next data member is m_configuration, which is also a pointer to a dynamically allocated object, but to the class ConfigAccess. This pointer represents global access to the configuration file system of the framework, which provides the ability to access user parameters globally. The Simulator class can freely pass around these pointers to other parts of the framework that require them for specific purposes.

The Logger class is a class that is meant to log, hold, and flush to a file specific messages during the framework's lifetime that are integral to helping the user understand what is going on. By making the Simulator class have the m_logger data member to reference this class, all of the logging of activity is done with respect and access to key moving parts of the framework.



(Figure 13: Activity Logger Output)

The logger class has unique functionality where it will timestamp all of the statements being logged and also flush each of the statements to a separate file for viewing. Since “m_logger” presently is available during the synchronous parts of the trading simulation, it is able to log many different statements depending on the context. Some of the more helpful instances of logging user activity are through catching errors or when trying to validate integral parameters for simulation.

```
std::string key = "REPORT_DATA_PATH";

std::vector<DateTime> ref_dates = m_dataAccess->Get_Ref_Date();
if (!m_configuration->Get_Param(key, report_directory)) {
    m_logger << "Could Not Get REPORT_DATA_PATH...CHECK CONFIGURATION FILE";

    return false;
}
```

(Figure 14: Parameter Validation Code-Snippet)

Here, while initiating type-safe access to parameters, the method can return false if the user has incorrectly formatted a parameter in their configuration file. By having the ability to log events like these, the Simulator class can not only stop the simulations when not all required validation is passed, but also provide to the user in a special log file what went wrong and where to mitigate it. Having a logger helped in debugging and building out the framework tremendously.

5.5.2 Flow of Simulation

The Simulator class is also responsible for controlling the flow of the trading simulations, whether that be backtests, optimizations, or WFA. Implementing this proved to be a challenge because of the different event structure in backtests, optimizations, and WFA. Backtests, optimizations, and WFA differ both specific outcomes after the simulation and the number of simulations being run. However, since a singular backtest is a singular trading simulation, the framework treats optimizations and WFA as many different units of backtests. The differences in implementation would be before and after the optimizations and WFA runs. The core event loop of backtests that is present in optimizations and WFA runs is split into the following parts:

1. Simulation-Required and User-Provided Parameters Initialization and Validation
2. Begin Simulation by Looping Through Trading Days
3. For Each Day, do the following:

a. Check Capital Constraints
    i. If any capital constraints are violated, end trading and generate reports
b. Check for and Process Existing Exit Orders
    i. If any exit orders are filled, close them
c. Check for and Process Existing Entry orders
    i. If any entry orders are filled, open them
d. Generate Entry and Exit Trading Signals based on new price data
e. Generate and Queue Orders based on Signals for the Next Day
f. Update all positions and portfolio metrics

4. End Simulation, Update Final Portfolio metrics, and Generate Reports

The whole point of treating backtests as repeatable units is that the framework does not have to implement the same logic for optimizations and WFA runs. Since optimization runs are a series of backtests with different parameters, and WFA runs are a combination of optimizations and backtests on specific separate data windows, the bulk of the trading simulation logic is the same across the board for the different types of simulations. The real differences are with the processing before and after the different types of simulations have occurred. Therefore, it is possible to allow the trading simulations in a backtest to be portable logic used elsewhere, repeatedly.

5.6 User-Model Interface

For the framework to be instantiated and run properly, one of the user-provided requirements is their own trading model. To accomplish this, the following rationale for implementation was followed:

1. Specific Functions/Methods for the User to define regarding trading strategy/rules in a trading model must be implemented
2. User-Provided Model must be a derived object of the Simulator class with access to specific methods
3. Example Model API for the user to follow and be a bridge

5.6.1 Required Functions for a Valid Trading Model

Whenever the simulator class runs any type of trading simulation, multiple methods are called to facilitate the placement of a trading model for the simulator class to use:

1. Virtual bool Apply_Model(InstrumentData* trading_block, DateTime day, Trading_Signals& trading_signals, std::map<std::string, std::string> params);
2. Virtual bool Generate_Long_Exit_Signal(InstrumentData* trading_block, DateTime day, std::map<std::string, std::string> params);
3. Virtual bool Generate_Short_Exit_Signal(InstrumentData* trading_block, DateTime day, std::map<std::string, std::string> params);
4. Add the stop orders and entry orders, and display config parameters and other stuff

Each of these methods represents what a trading model is from the framework's perspective, as these methods are each called during trading simulations. For the user-provided trading model to be a valid and usable trading model by the framework's standards, the user-provided model must adhere to the requirement of defining these functions. Each of the functions required for a user-provided trading model utilizes the "virtual" keyword in its declaration. A virtual function indicates that at runtime, the exact definition of these functions used is that of the object calling it. For example, whenever the framework looks to apply specified calculations and check for

entry signals, the framework will invoke the "Apply_Model" method, but look for the user's definition of these functions. The virtual keyword specified in the function prototype by the Simulator class allows the framework to achieve runtime polymorphism and look for the user-provided model's definition of these functions at runtime instead of the Simulator class.

5.6.2 Derived Object of Simulator Class

For the user-provided model to work, the framework requires that the user-provided model be a derived class of the Simulator class. This is done for the following reasons:

1. The required functions for entering, exiting, and managing positions are virtual functions
2. The Simulator class holds access to parameters and data, and the ability to initiate trading simulations

By making the user-provided model a derived class, it is able to override the definitions of the functions required to run and evaluate a trading model with its own definitions at runtime. Furthermore, since the Simulator class holds access to all of the data and parameters required, the user-provided class can inherit all the public accessor methods required to apply calculations and check for entry or exit signals, or to get specific parameters, or to run specific forms of simulation. This effectively allows the framework to plug in user models where all the development needed by the user is applying calculations, checking for entry/exit signals, and customizing their entry and exit orders.

5.6.3 Example Model API

To allow for a more concise blueprint to follow, the user-provided model will be a derived object of the L_S_MODEL_EXAMPLE class, which itself is a derived object of the

Simulator class. The reason for this is to allow abstraction of all the more complex parts of the framework, the user does not need to edit and provide clear instructions on what needs to be defined by the user to start simulating their trading model. Thus, the class "L_S_MODEL_EXAMPLE" is provided to help the user interface with the framework.

The User-Provided model is a derived class of the L_S_MODEL_EXAMPLE class, and the L_S_MODEL_EXAMPLE class is a derived class of the Simulator class. The User-Provided Model is meant to be the key for the framework to work, but the L_S_MODEL_EXAMPLE class is meant to be the blueprint for the minimum requirements of the user-provided model.

By allowing the user-provided model to follow the blueprint of the L_S_MODEL_EXAMPLE class, the user will always know what methods their model class needs to define to run the framework.

```
L_S_MODEL_EXAMPLE engine;
engine.Load_Config_File(configFilePath);
engine.Load_Data_Files();
engine.Simulate_Backtest(1);
```

(Figure 15: Main File Event Loop)

5.7 Optimization Implementation

Optimizations and WFA are similar to backtests, but provide results different from those of a normal backtest. In a backtest, the user of the framework will receive reports on their model's performance in terms of trades, backtest metrics, and time-based reports.

Parameterization

In optimization, the goal is to run many different backtests, but with different parameters, so the user can see which parameters best allow their model to perform. Optimization requires the user to specify their specific parameters, and from there, subsets with all the different parameter combinations are created. The “Optimization” class handles the following tasks related to optimization:

1. Storing User-Specified Fitness Function
2. Adding and Storing User-Specified Parameters Focused on for Parameterization
3. Creating Parameter Subsets
4. Storing Parameter Backtest Results
5. Generating Optimization Reports

5.7.1 User-Specified Fitness Function

Optimization runs need some type of metric to be evaluated against to see which parameter combinations did better after all the trading simulations are done. The user must specify a fitness function in their configuration file, which will dictate what performance metric to use to evaluate all the optimization runs. The user can specify the following:

1. Total Trades
2. Net Profit
3. Gross Profit
4. Gross Loss
5. Number of Winning Trades
6. Number of Losing Trades
7. Max Winning Trade
8. Max Losing Trade

9. Average Winning Trade
10. Average Losing Trade
11. Win/Loss Ratio
12. Expectancy Score
13. TS Index

The listed metrics are all part of a singular backtest's metrics reports, which will generate a report on a given model's performance during a singular backtest.

5.7.2 Adding and Storing User-Specified Parameters Focused on for Parameterization

```
# Parameters to Optimize during WFA
qqq_slow_length=[50,100,200]
qqq_fast_length=[25,50,100]
slow_length=[100,50,200]
fast_length=[25,50,100]
```

(Figure 16: Parameters to Optimize)

The user has the ability to add parameters for optimization in a configuration file where the parameters are listed with different values as an array. Parameters added in the configuration file is generally the way to do this as like with all other parameters, everything is separated from the actual application logic.

5.7.3 Creating Parameter Subsets

When the user has added all of their different parameters for optimization, the "Optimization" class will shuffle all the different values for each class, and create subsets that

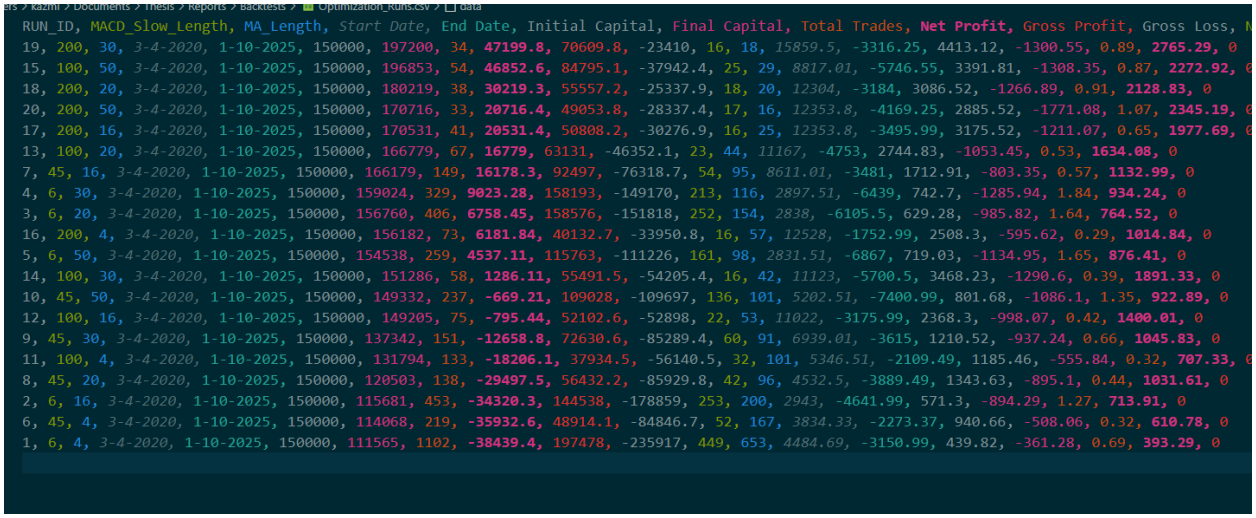
represent every combination of each of the specified parameters, where each parameter will have 1 different value in a subset. From here, the parameter combinations are numbered and stored in a map data structure. Each key for the map is the numbered parameter combination, and the individual subsets are stored per that integer combination key.

5.7.4 Storing Parameter Backtest Results

Since each optimization run is unique due to having different parameters, each run's backtest metrics results are synthesized into a "Backtest_Results" object. Backtest results represent a singular run's performance across all the backtest metrics in a normal backtest report. To associate each parameter combination with the correct backtest_results object, each backtest_results object is stored in a separate map similar to the parameter subsets. The key for this map is an integer key that represents a parameter subset's numbered combination, and the backtest results corresponding to that specific parameter subset.

5.7.5 Generating Optimization Reports

When optimization runs are done, the report that needs to be generated is different from the normal reports generated by a backtest. Optimization runs use the fitness function to measure the performance of individual runs with different parameters. The goal is to find out the best set of parameters that yields the best possible performance, to then explore in further model development. When synthesizing an optimization report, the framework will generate one as follows:



RUN_ID	MACD_Slow_Length	MA_Length	Start Date	End Date	Initial Capital	Final Capital	Total Trades	Net Profit	Gross Profit	Gross Loss	N
19	200	30	3-4-2020	1-10-2025	150000	197200	34	47199.8	70609.8	-23410	16	18	15859.5	-3316.25	4413.12	-1300.55	0.89	2765.29	0
15	100	50	3-4-2020	1-10-2025	150000	196853	54	46852.6	84795.1	-37942.4	25	29	8817.01	-5746.55	3391.81	-1308.35	0.87	2272.92	0
18	200	20	3-4-2020	1-10-2025	150000	180219	38	30219.3	55557.2	-25337.9	18	20	12304	-3184	3086.52	-1266.89	0.91	2128.83	0
20	200	50	3-4-2020	1-10-2025	150000	170716	33	20716.4	49053.8	-28337.4	17	16	12353.8	-4169.25	2885.52	-1771.08	1.07	2345.19	0
17	200	16	3-4-2020	1-10-2025	150000	170531	41	20531.4	50808.2	-30276.9	16	25	12353.8	-3495.99	3175.52	-1211.07	0.65	1977.69	0
13	100	20	3-4-2020	1-10-2025	150000	166779	67	16779	63131	-46352.1	23	44	11167	-4753	2744.83	-1053.45	0.53	1634.08	0
7	45	16	3-4-2020	1-10-2025	150000	166179	149	16178.3	92497	-76318.7	54	95	8611.01	-3481	1712.91	-803.35	0.57	1132.99	0
4	6	30	3-4-2020	1-10-2025	150000	159024	329	9023.28	158193	-149170	213	116	2897.51	-6439	742.7	-1285.94	1.84	934.24	0
3	6	20	3-4-2020	1-10-2025	150000	156760	406	6758.45	158576	-151818	252	154	2838	-6105.5	629.28	-985.82	1.64	764.52	0
16	200	4	3-4-2020	1-10-2025	150000	156182	73	6181.84	40132.7	-33950.8	16	57	12528	-1752.99	2508.3	-595.62	0.29	1014.84	0
5	6	50	3-4-2020	1-10-2025	150000	154538	259	4537.11	115763	-111226	161	98	2831.51	-6867	719.03	-1134.95	1.65	876.41	0
14	100	30	3-4-2020	1-10-2025	150000	151286	58	1286.11	55491.5	-54205.4	16	42	11123	-5700.5	3468.23	-1290.6	0.39	1891.33	0
10	45	50	3-4-2020	1-10-2025	150000	149332	237	-669.21	109028	-109697	136	101	5202.51	-7400.99	801.68	-1086.1	1.35	922.89	0
12	100	16	3-4-2020	1-10-2025	150000	149205	75	-795.44	52102.6	-52898	22	53	11022	-3175.99	2368.3	-998.07	0.42	1400.01	0
9	45	30	3-4-2020	1-10-2025	150000	137342	151	-12658.8	72630.6	-85289.4	60	91	6939.01	-3615	1210.52	-937.24	0.66	1045.83	0
11	100	4	3-4-2020	1-10-2025	150000	131794	133	-18206.1	37934.5	-56140.5	32	101	5346.51	-2109.49	1185.46	-555.84	0.32	707.33	0
8	45	20	3-4-2020	1-10-2025	150000	120503	138	-29497.5	56432.2	-85929.8	42	96	4532.5	-3889.49	1343.63	-895.1	0.44	1031.61	0
2	6	16	3-4-2020	1-10-2025	150000	115681	453	-34320.3	144538	-178859	253	200	2943	-4641.99	571.3	-894.29	1.27	713.91	0
6	45	4	3-4-2020	1-10-2025	150000	114068	219	-35932.6	48914.1	-84846.7	52	167	3834.33	-2273.37	940.66	-508.06	0.32	610.78	0
1	6	4	3-4-2020	1-10-2025	150000	111565	1102	-38439.4	197478	-235917	449	653	4484.69	-3150.99	439.82	-361.28	0.69	393.29	0

(Table 4: Parameter Optimization Report)

The report will sort all runs from highest to lowest per the user-specified fitness function. Each run will be reported with three parts comprising it:

1. The run-id, which is the key that links the map of parameter subsets and the map of backtest results objects together
2. The individual parameter values for each of the parameters tested
3. The backtest metrics results are stored in each Backtest_Results object for a given run

By having the report generated in this format, the user can see which iteration of their trading model did the best per their use-specified fitness function, but also analyze the results of all runs as needed.

5.7.6 Concurrency in Optimizations

One very large problem in algorithmic trading is how long numerous runs can take. Oftentimes, the number of optimizations run can be exponentially longer than intended due to the many different values for a given parameter that need to be tested. Having many different

parameters can turn optimizing trading models into high-dimensional optimization problems where a large number of variables are being tested. To mitigate this, the framework implements asynchronous multi-threaded optimization runs.

By running multi-threaded optimization runs, the framework is able to substantially lower the amount of time required to wait for trading model optimization to finish. If the framework chose to maintain a synchronous loop for the optimizations, that would mean waiting for each optimization run to finish before starting another, which is very arduous. Since each optimization run is run with different parameters, each run is mutually exclusive in terms of results and performance.

One problem with running processes in multi-threading is when a data source shared by multiple processes has to be written to or modified. How each simulator works is that the data and configuration files that are accessible during a run are only ever read from, never written to. Each run will share the required parameters, such as Start Date or End Date, or use the same price data for each of the stocks; however, each run will purposely never modify any existing data that is used in multi-threaded runs. The data access and configuration access in the framework are done through the use of pointers to a dynamically allocated object, so there is a global access point that the framework can provide to any of the moving parts.

To implement concurrency within optimization for this framework, three major components were used: std::async, std::future, and std::launch. Each of these components represents different aspects of asynchronous and concurrent programming. Std::async allows the initiation of function calls to be asynchronous and promises a result, not knowing when the result will be finished. By being able to launch multiple optimization runs in different contexts, the execution becomes asynchronous. Std::async by itself does not always open a new thread for

function calls it initiates, and therefore it is not always multi-threading. Std::future allows the ability to wait on promises from std::async and is used to tell threads to wait and retrieve the values from this specific asynchronous function. While std::async and std::future handle the asynchronous part of optimization, a third component is used to ensure that function calls made by std::async always result in a new thread when available. Std::launch is meant to specify specific policies for std::async whenever tasks are initiated. By using std::launch::async, the framework will be able to dictate that asynchronous optimizations are run on a new thread where possible, without the fear of overscheduling.

By adopting this three-part design for the concurrency implementation, the framework is able to ensure asynchronous optimizations to dramatically reduce optimization run completion times. Each optimization run's numerous simulations are able to be run multi-threaded when possible, without having to deal with race conditions, complex management of resource consumption, or thread deadlocks of resources.

5.8 Walk-Forward-Analysis

WFA (Walk-Forward Analysis) is a special type of optimization technique that requires a combination of optimization and backtests. Since the backtests in the framework are treated as a reusable unit, and the optimizations are a series of backtests, then that means WFA is also a series of backtests in its own right; however, WFA, like optimization, requires a special implementation regarding preparing for the simulation and processing of results.

5.8.1 Why Choose WFA

WFA is one form of trading model validation and while there are different methods to analyze a trading model's performance, WFA is definitely one of the better methods. Some of the other methods which are still effective and can be used in conjunction with WFA are "Monte Carlo Simulations" and normal out-of-sample testing.

Monte Carlo Simulations are meant to look at historical returns of a model and randomly resample them through many different simulations to create a large distribution of values where each value is your models performance over a given simulation. A uniform value per your model's performance can be observed via many historical returns. This uniform value can help provide a statistically valid outlook on your model's average or expected performance through a probabilistic distribution of runs. This can definitely be helpful, however it is important to have out-of-sample data used to test your model which monte-carlo does not include. It takes your model's historical returns and "randomly" resamples the returns and trades to create different return scenarios for your model which is not using out-of-sample data to evaluate your model. It can be effective but it does not focus on the right methods for trading model evaluation which would be testing on consistent segments of unseen, real data.

Another method of trading model evaluation is "out-of-sample" testing where out-of-sample data is used to test the model. This is built directly into WFA as each round of WFA has an optimization run on in-sample data with the best parameter combinations used by the model when it is back tested on multiple out-of-sample data windows.

Overall, it is key to understand there are many different tools for trading model evaluation, however WFA is considered one of better tools for this task due to its nature of evaluating models on different segments of out-of-sample data.

5.8.2 Anchored WFA

Anchored WFA is the form of WFA implemented, where the in-sample training window each round of testing gets longer by adding the out-of-sample data window tested previously, and moving the out-of-sample data window to the next same-sized range of trading days. There are four parameters required for walk-forward analysis by the framework that must be written in the configuration file:

1. IS_START → Start of In-Sample Data Window
2. IS_END → End of In-Sample Data Window
3. OS_START → Start of Out-of-Sample Data Window
4. OS_END → End of Out-of-Sample Data Window

Each of these dates is integral to the construction of the required data windows to run WFA. The first round of WFA will be running with the data windows at these dates; however, starting the next round, both of the windows will change.

Each round of WFA will consist of concurrent optimization runs of the user-provided model, and per the user's fitness function, the best set of parameters from the optimization run will be used to backtest the model on the out-of-sample window and record the results.

Finally, after the WFA runs have been concluded, the user will be provided a WFA Report, which will show all the backtest results for the user's provided model on the different out-of-sample trading windows.

5.8.3 WFA Date Window Construction

Before WFA can be run with the user-provided trading models, the date windows for WFA must be constructed. These specific date windows will allow the framework to figure out

which dates will be run with optimization and which dates will be run with a backtest for a given round, respectively.

Firstly, the framework will find the size of the first out-of-sample data window in terms of valid trading days. The framework will then use that number of days to create a new out-of-sample data window each round. Furthermore, the in-sample data window will be appended with the out-of-sample data window so that the end of every in-sample data window is the beginning of the new out-of-sample data window for that round. A new out-of-sample data window will be either the size of the original out-of-sample data window or a lesser value, indicating that the end of the trading dates has been reached.

Once the four required dates are ready and the size of the out-of-sample window is calculated, the framework will create subsets of dates representing each WFA round. Each of these respective rounds will represent a large in-sample data window and a new out-of-sample data window when compared to the prior round's data windows, respectively. Each subset will consist of DateTime objects representing the required WFA dates for that round and will be stored in a specific map object representing that subset/WFA round. Each map representing the date windows for the round of WFA is then stored in a vector.

It is important to understand however that each window can be of a different size. So, in WFA, there is a moving in-sample range as well as the expected moving out-of-sample date range; this split is generally done as 70% of the entire range is in-sample optimization, and 30% is out-of-sample backtesting. However, the framework allows specific dates to be picked for the initial WFA window construction and because of this, the split is up to the user. The in-sample date window in anchored WFA grows each round as the previous out-of-sample date range is added to it and so it does not make sense to adhere to specific % splits as each case is different

and up to the specified settings in the configuration files for the initial WFA date window dates and the length of the time-series stock data provided.

5.8.4 Training and Testing on Data Windows

For each round of WFA, the framework will run optimization on the in-sample window and use those results to then run a singular backtest on the out-of-sample window.

When optimization is run on the in-sample dataset, it is a series of concurrent runs to yield the best parameters for that in-sample dataset. Each optimization run will yield an object of the class "Backtest_Results," and once all runs are completed, the optimization class will record the parameters that yielded the best results per the user-metric.

Once the best parameters from the in-sample training window are found, those same parameters will be applied to the model, and the model will then be backtested over the out-of-sample data window. The goal is to understand how the model performs on "unseen" data, and by backtesting the model on different out-of-sample data, the performance results yielded will be closer to live performance. After each backtest on the out-of-sample window, a "Backtest_Results" object containing the results is then handed off to the "Optimization" class

The framework being able to capitalize on the portability of the code for backtests, whether in optimizations or a normal backtest, is what allows the WFA to be implemented rather smoothly.

5.8.5 WFA Report

When WFA is finished, a report on the performance of the model during the different out-of-sample windows is generated. This report is similar to the backtest metrics report and

parameters report, which will consolidate information on the model's performance for each out-of-sample data window.

Using the same approach as optimization, where each method returns a backtest results object, each out-of-sample data window's backtest will return the same object, and the optimization class will store each of those objects after each round. When WFA has completely finished, the optimization class will then generate a report for the WFA run.

Chapter 6

Framework Demonstration

Once the user has provided the necessary components to the framework, the framework will then be able to run backtests or WFA to generate data on model performance from reports after simulations.

In this section, an example model will be provided to showcase the capabilities of the framework. The framework will run the model through many trading simulations and generate data in the form of performance reports. These performance reports will then be visualized in python to then extrapolate meaningful insights on the provided trading model's capabilities and shortcomings.

6.1 Loading Configuration Files and Data Files

Once the user has provided the proper configuration files and financial data per the formatting rules and the user provided model; they will have to create a main.cpp file such as this. In this file, the user's provided model is imported and instantiated, and a string filepath to the user's configuration file is also stored in a string variable. The user must then load the configuration file, followed by loading their data files and then run their choice of a backtest or WFA.

```cpp
#include"Example_Model_1.h"
int main(int argc, char** argv)
{
    std::string configFilePath = "C:/Users/kazmi/Documents/Thesis/main_configuration.toml";
    Example_Model_1 engine;

    engine.Load_Config_File(configFilePath);
    engine.Load_Data_Files();
    engine.RUN_WFA();
```

(Figure 17: Main File Code-Snippet)

```
Loading Configuration File at C:/Users/kazmi/Documents/Thesis/main_configuration.toml...
Sucessfully Loaded Configuration File...
Loading Configuration File at C:/Users/kazmi/Documents/Thesis/parameters.toml...
Sucessfully Loaded Configuration File...
[2025-04-13 19:03:37] Logger has been successsfully created!
[2025-04-13 19:03:37] Accessed the Configuration Map via: HIST_DATE_REF_TICKER Key and Set the reference ticker to: AMZN
[2025-04-13 19:03:37] Successfully added the parameter: qqq_slow_length to the optimization pool
[2025-04-13 19:03:37] Successfully added the parameter: qqq_fast_length to the optimization pool
[2025-04-13 19:03:37] Successfully added the parameter: slow_length to the optimization pool
[2025-04-13 19:03:37] Successfully added the parameter: fast_length to the optimization pool
[2025-04-13 19:03:37] Loading Data Files Directory...
[2025-04-13 19:03:37] Loaded and Inserted AAPL Data Block
[2025-04-13 19:03:37] Loaded and Inserted ADBE Data Block
[2025-04-13 19:03:37] Loaded and Inserted AMZN Data Block
[2025-04-13 19:03:37] Loaded and Inserted AVGO Data Block
[2025-04-13 19:03:37] Loaded and Inserted CDNS Data Block
[2025-04-13 19:03:38] Loaded and Inserted COST Data Block
[2025-04-13 19:03:38] Loaded and Inserted GILD Data Block
[2025-04-13 19:03:38] Loaded and Inserted GOOGL Data Block
[2025-04-13 19:03:38] Loaded and Inserted MSFT Data Block
[2025-04-13 19:03:38] Loaded and Inserted NFLX Data Block
[2025-04-13 19:03:38] Loaded and Inserted NVDA Data Block
[2025-04-13 19:03:38] Loaded and Inserted PANW Data Block
[2025-04-13 19:03:38] Loaded and Inserted QQQ Data Block
[2025-04-13 19:03:38] Loaded and Inserted SBUX Data Block
[2025-04-13 19:03:38] Loaded and Inserted TMUS Data Block
[2025-04-13 19:03:38] Loaded and Inserted TSLA Data Block
[2025-04-13 19:03:38] Loaded and Inserted TXN Data Block
[2025-04-13 19:03:38] Done Loading Data Files
[2025-04-13 19:03:38] Initializing Model with Predefined Calculations...
[2025-04-13 19:03:38] Applied Predefined Calculations to AAPL
[2025-04-13 19:03:38] Applied Predefined Calculations to ADBE
[2025-04-13 19:03:38] Applied Predefined Calculations to AMZN
[2025-04-13 19:03:38] Applied Predefined Calculations to AVGO
[2025-04-13 19:03:38] Applied Predefined Calculations to CDNS
[2025-04-13 19:03:38] Applied Predefined Calculations to COST
[2025-04-13 19:03:38] Applied Predefined Calculations to GILD
[2025-04-13 19:03:38] Applied Predefined Calculations to GOOGL
[2025-04-13 19:03:38] Applied Predefined Calculations to MSFT
[2025-04-13 19:03:38] Applied Predefined Calculations to NFLX
[2025-04-13 19:03:38] Applied Predefined Calculations to NVDA
[2025-04-13 19:03:38] Applied Predefined Calculations to PANW
[2025-04-13 19:03:38] Applied Predefined Calculations to QQQ
[2025-04-13 19:03:38] Applied Predefined Calculations to SBUX
[2025-04-13 19:03:38] Applied Predefined Calculations to TMUS
[2025-04-13 19:03:38] Applied Predefined Calculations to TSLA
[2025-04-13 19:03:38] Applied Predefined Calculations to TXN
[2025-04-13 19:03:38] Done Initializing Model with Predefined Calculations
[2025-04-13 19:03:38] Checking WFA Date Ranges...
[2025-04-13 19:03:38] Done checking WFA Date Ranges...
```

(Figure 18: WFA Log Output)

The framework will open a command prompt screen with example output logging the activity during the lifecycle of this run. This indicates that the framework has successfully loaded the configuration and data files and applied any predefined calculations per the user's model to the data blocks representing each stock.

6.2 Running WFA or a Backtest

Once the user has successfully loaded in their configuration and data files, they must choose to either run a backtest or WFA.

```
#include"Example_Model_1.h"
int main(int argc, char** argv)
{
    std::string configFilePath = "
    Example_Model_1 engine;

    engine.Load_Config_File(configF
    engine.Load_Data_Files();

    engine.RUN_WFA();
    //engine.Simulate_Backtest();
```

(Figure 19: WFA Main Code Snippet)

The method chosen must be specified by the user and is as simple as only choosing one of these two methods. Because the framework allows customization of the settings to be abstracted from the application logic via configuration files, and the calculations for all simulations to be done from a user-provided model, the framework is able to dramatically streamline the process of initiating and customizing trading simulations.

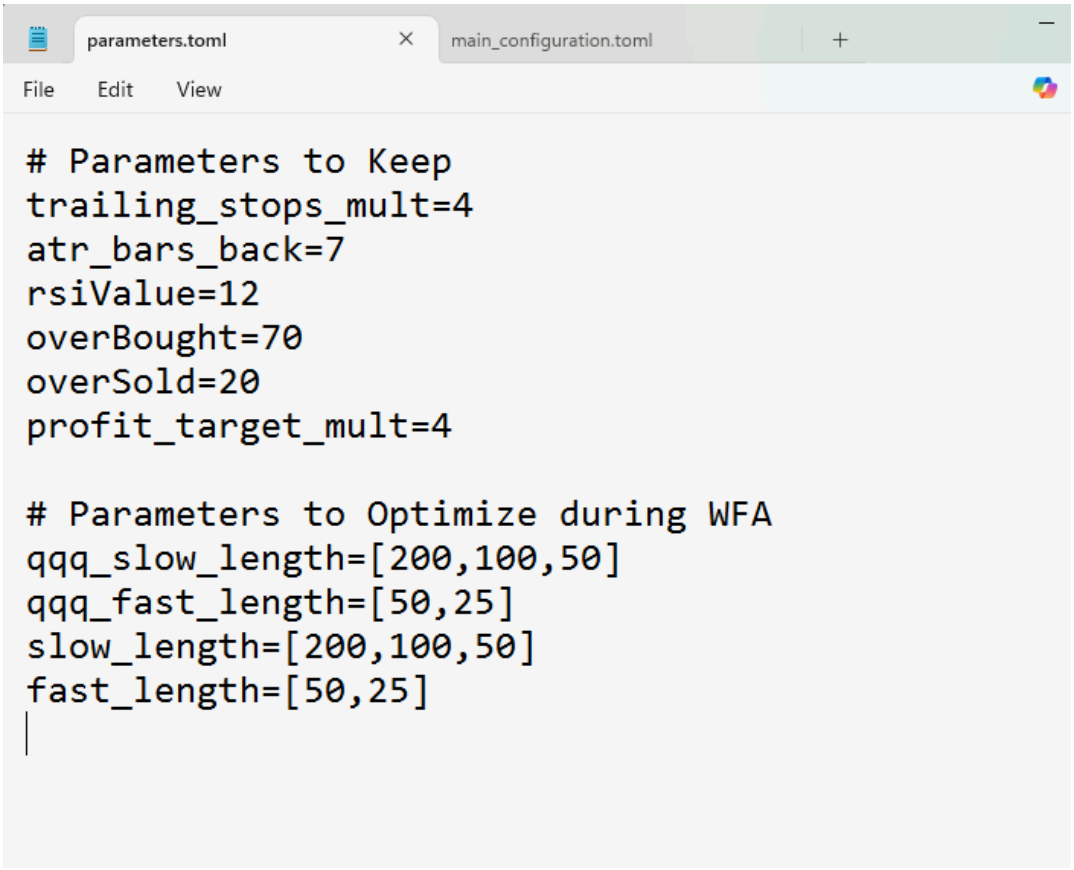
6.3 User-Provided Model Chosen

A specific trading model is provided for the sake of demoing the framework's report generation capabilities after running WFA and backtests. The model provided uses a mixture of calculations and parameters to create multiple entry and exit positions so the framework can show off the optimization and WFA performance of the model.

The entry is based upon the RSI of the stock, the 50 and 200 Day Moving Averages of the stock's weighted price, and the 50 and 200 day moving average of the QQQ's VWAP. The model will use two combinations of these three components to generate entry signals into long and short positions.

The exit is based upon either one of two conditions: the stock's 50 day moving average crossing above/below its 200 day moving average, or the stock's RSI crossing above/below its overbought/oversold levels.

For the sake of managing risk, the model will also enable profit targets by allowing a multiple scalar value applied to its calculated Average True Range. Once the price of the stock hits this number generated, the position's profit target will be received and the position will close with a limit exit order.



(Figure 20: Parameters Optimization Configuration File)

The following parameters will be optimized during the WFA for this model and then the best combinations will be explored via separate backtests to provide a comparative analysis on the model's performance.

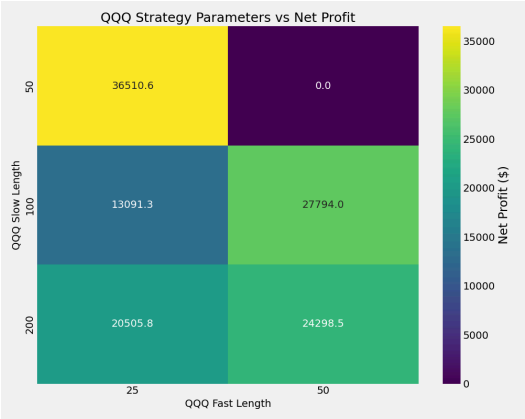
6.4 Using Reports Generated by the Framework

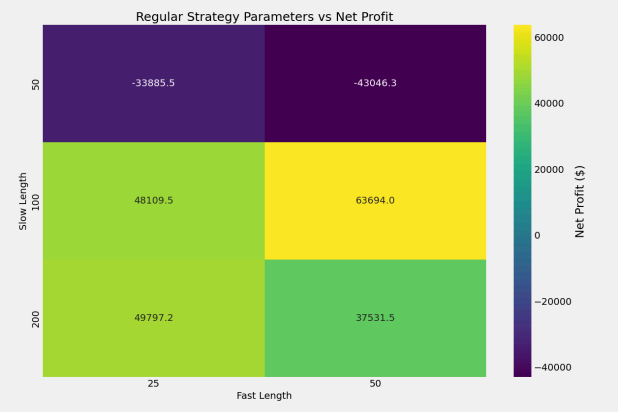
Once the WFA is done, further backtests will be done to then examine the model's performance with the best set of parameters from the WFA. Once both runs are done, the analysis of the model's performance can begin via report visualizations.

6.4.1 Optimization Visualizations

There were four parameters tested in the in-sample optimization runs during WFA each with their own distinct values. An optimization report was generated after this run which showcases the performance of each combination of parameters per an optimization run.

When analyzing optimization runs, the relationship of the parameters to one another and the specified user function is key as this can show how the parameters work together during a model's simulation. To visualize this aspect, heatmaps were the type of visualization chosen to visualize these parameter relationships and their outcome on model performance.

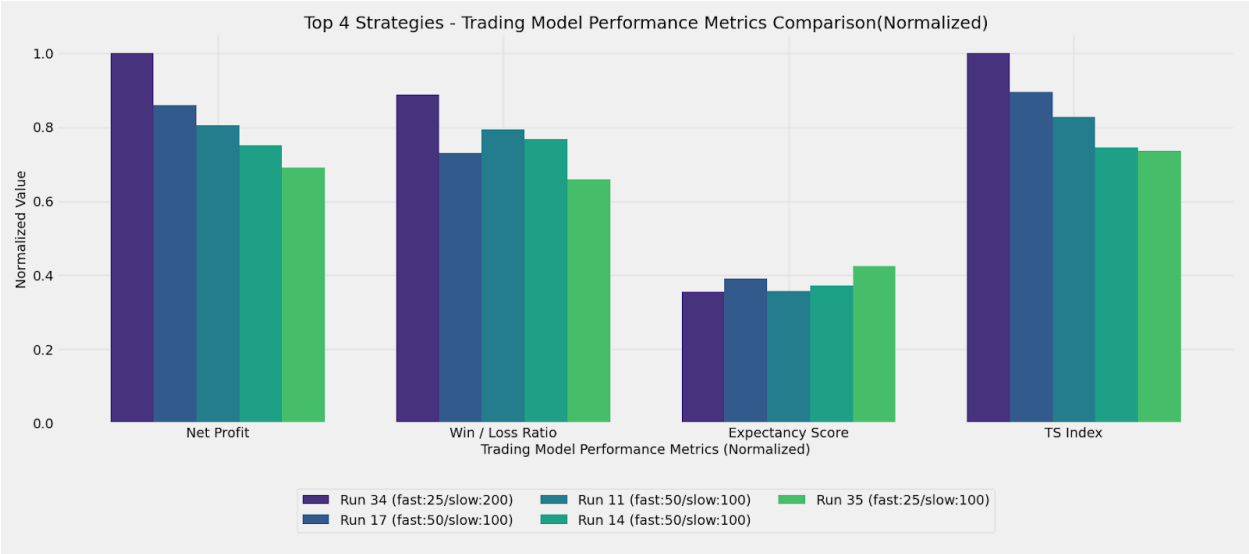




(Figure 21: QQQ Parameters Heatmap) (Figure 22: Equity Parameters Heatmap)

Here we have all four sets of parameters being compared with their counterpart along with a varying color map of the user-specified fitness function. Here we can observe how the 50 and 25 parameter combinations for the QQQ Slow Length and QQQ Fast Length, respectively generated better performance than their other combinations. Furthermore, the values 50 and 100 for the fast length and slow length also provided substantial performance over their other parameter counterparts. Now, this does not necessarily mean that they should be chosen, but rather, it provides more insight on which area of values to concentrate on for this model's parameters and performance.

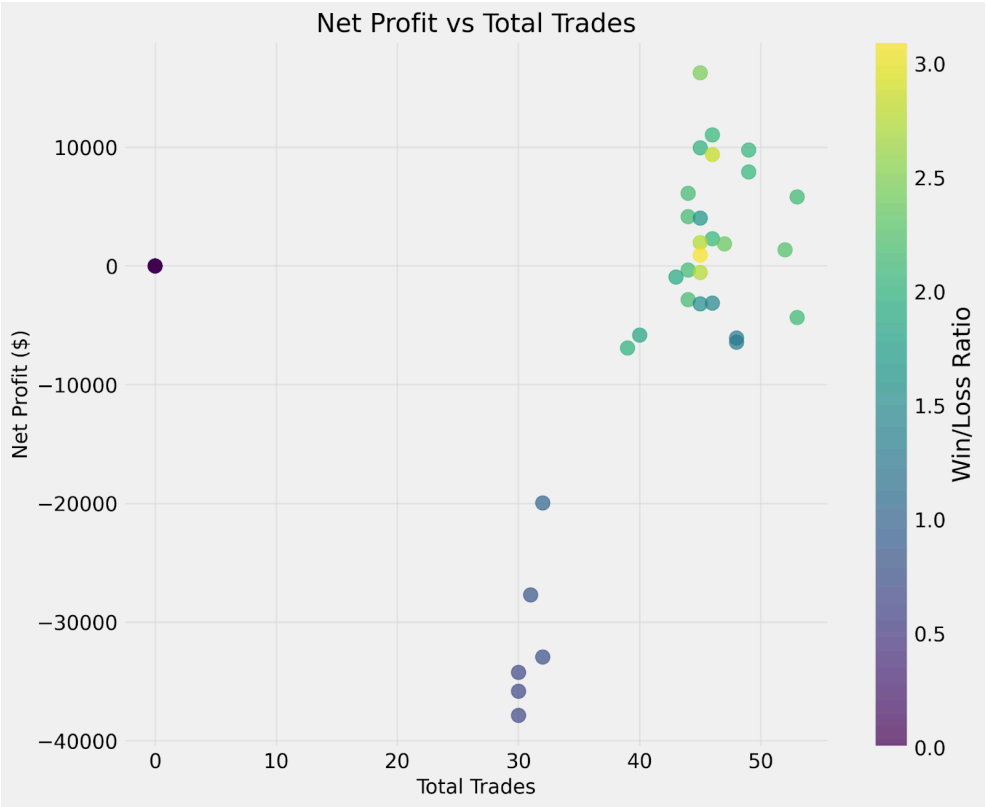
Another key aspect of optimization is looking at how specific runs did in regard to lesser obvious metrics then something like net profit. Since each optimization run has its own run id that represents its parameters, it is simple to then graph and visualize these metrics.



(Figure 23: Trading Model Simulation Metrics, Normalized)

In this example, these specific metrics are normalized to be adjusted to this graph and then visualized in tandem with values of the same metric but from different runs. This graph provides key comparative analysis of the metrics across the top 4 parameter combinations to allow a more meaningful view of the model's performance.

While it is true that Run 34 had a much higher net profit then the other runs, it had much higher values for metrics like WIn/Loss Ratio and the TS Index. Therefore, the following statement can be extrapolated: Run 34's model parameters took on trades with a higher profit upside and also more efficiency when compared to the other runs which were less profitable but also had lesser metrics



(Figure 24: Net Profit vs Total Trades Metrics Dotplot)

This figure outlines the relationship of the net profit and total trades made by each of the optimization runs with the win/loss ratio as a color scale. The figure shows that the lower number of trades made lead to substantially negative profits while more trades in the 40 to 50 trademarks lead to substantially higher trades with a higher win/loss ratio. While this figure does not give the user's the best parameter values for their model, it shows the model's widely varying performance across differing sets of parameters. If a model was optimized with many different parameters values but all the points in this figure were in the negative profit range while the total trades increased or decreased, it can be reasonably deduced that that specific model is not robust no matter how many different parameter values. Here the figure shows that with the right parameter values, the model's performance can vary drastically. A key extrapolation of a robust trading model from this figure would be editing the entry and exits such that the bottom

performing runs of the model are above 0. This would indicate a model's entry logic being so robust that it is consistently profitable despite increasing the number of total trades.

6.4.2 WFA Visualizations

The foremost goal of WFA is not necessarily to provide the model with the most optimal parameters, but rather test if the most optimized version of the model can work well when dealing with unseen data. WFA brings algorithmic model development one step closer to live deployment and trading since both situations involve the already-trained trading model encountering data it has not seen and needs to evaluate to enter and exit trading positions.

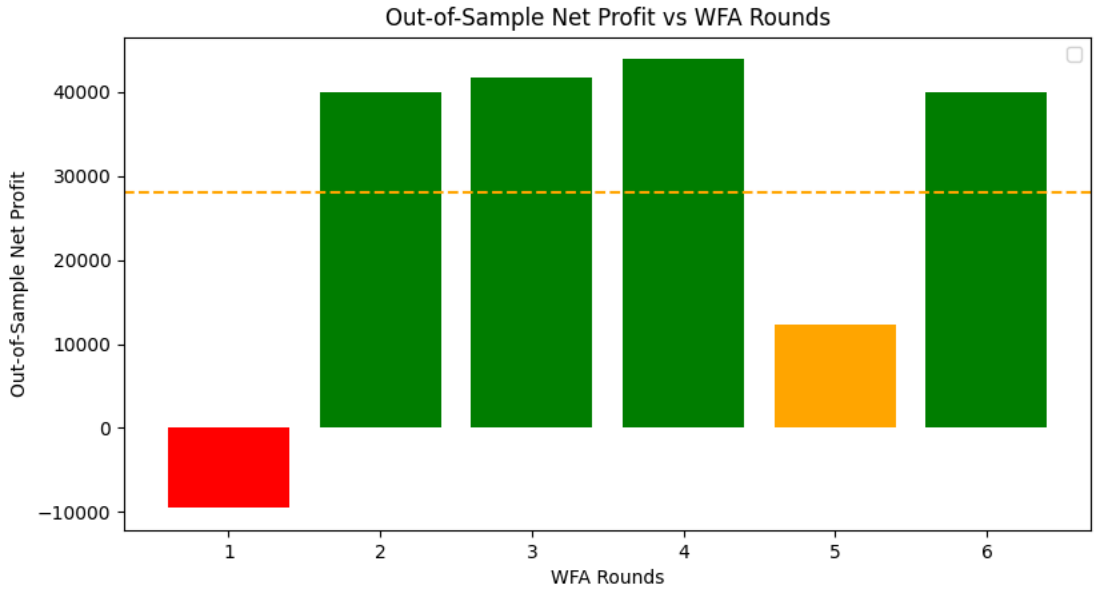
To analyze this type of performance, a special metric must be visualized called the W.F.E. score which stands for walk-forward efficiency. Here, the ratio of the in-sample and out-of-sample's net profit is taken with the higher the ratio in terms of % the more robust the model was during that out-of-sample period.

Walk-Forward-Efficiency Perfomance Analysis

Round	RUN_ID	In-Sample Net Profit	Net Profit	W.F.E Score	Total Trades	Win / Loss Ratio	Out-of-Sample Net Profit	Win/Loss Ratio
1	35	$37,508.80	-9531.04	-25.41%	34	0.89	$-9,531.04	0.89
2	35	$37,522.90	39979.8	106.55%	25	3.17	$39,979.80	3.17
3	29	$52,210.60	41720.0	79.91%	59	2.69	$41,720.00	2.69
4	17	$75,819.40	43865.9	57.86%	47	2.62	$43,865.90	2.62
5	6	$14,978.50	12374.4	82.61%	4	3.01	$12,374.40	3.01
6	34	$107,429.00	40015.1	37.25%	42	1.63	$40,015.10	1.63

(Table 1: WFE Visualization)

The W.F.E. score is key in making sure the WFA for a trading model is consistently good as this can indicate a robust trading model. However, as seen in round 1 where the W.F.E. was a negative value, and round 6th where the WFE score fell off dramatically, there is cause for concern on this model's performance in terms of recent data. This does not mean that the model and its parameters should be thrown out, but rather these out-of-sample periods should be investigated and retested. The Total Trades metrics for each round is also key in making sure the model is robust because you want a consistent number of trades for a model in each out-of-sample period. The fact that round 6 had 42 trades but only 37.25% for the W.F.E. score,round 1 had 34 trades but a negative W.F.E. score, and round 5 had a W.F.E. score of 82.61% but only 4 trades is definitive proof for investigation into those date ranges and more testing with this model and its trade criteria.



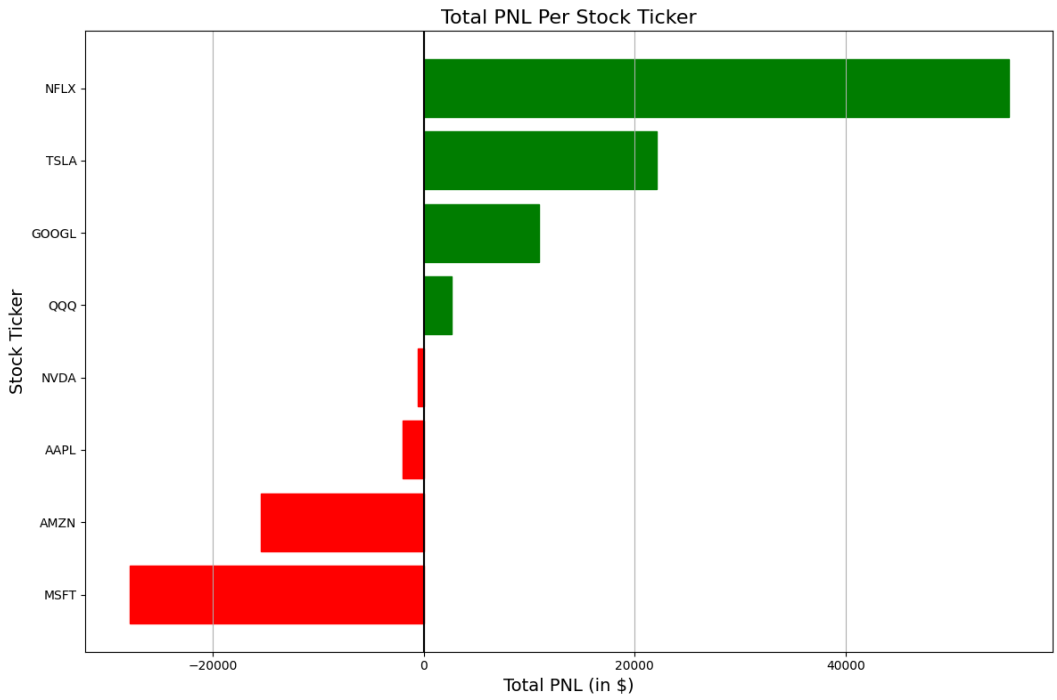
(Figure 25: WFA Runs)

The goal of a trading model is to make consistent profits while minimizing loss and risk. In the second figure, we can definitely see the performance of the model during its out-of-sample rounds be represented differently from its W.F.E. scores in the prior visualizations. The dotted line represents the mean net profit of the model's net profit from all of the out-of-sample periods. Round 6 in the prior figure had a W.F.E. score of 37.25% yet its net profit is well above the mean per this figure. Despite the W.F.E score being lower for round 6, the fact that the model was able to perform well and make a stable net profit well above the mean is a good sign of a robust trading model.

These two figures are able to provide visualizations of the trading model's performance through different aspects through the entire out-of-sample periods and show key insights into the remedying of a model that is over-fitted.

6.4.3 Trades List Visualizations

Besides the parameters and rules for entering and exiting trades, the trading strategy behind a model's trading logic also includes the list of stocks in its portfolio. When a backtest is simulated, one of the reports generated is the tradeslist report. This report will look at all the trades by each security. Analyzing and visualizing this report helps to understand under and over performers in a portfolio.



(Figure 26: Tradeslist Visualization)

In this figure, the stocks used in the portfolio are visualized by their total net profit from a backtest that uses the best parameters from the previous WFA. The visualization shows the net profit of each of the stocks being positive except NVDA,AAPL,AMZN, and MSFT. Now since this is only one backtest; it is not necessarily proof for removing NVDA, AAPL, AMZN, and MSFT from the portfolio as there are different variables that could have led to the poor performance from this trading model with these stocks. However, this highlights a need for investigation into the relationship of the model's performance with respect to these stocks. This could mean backtesting with different parameter combinations to see if these stocks are still detrimental to the model's performance. By analyzing the performance of the model by stock ticker, returns can be dramatically improved by removing under-performers and either adding a higher position size to over-performers or adding different stocks to the portfolio.

Chapter 7

Conclusion

7.1 Framework Goals

With the substantial rise of algorithmic trading and the increased availability of stock investments, there is an expanding need for software tools to facilitate methodical and accurate trading model development. This framework is meant to provide the proper validation methods for investors looking to create robust trading models to produce financial profits from investing. The flexibility of this framework in allowing users to provide their own trading models, customize settings via configuration files, and generate performance data. This allows investors to extrapolate meaningful insights on their trading model's performance.

7.2 Challenges in Design and Development

While the framework was developed properly per expectations, there were some challenges faced in development. This project entailed a lot of time in development as the needs for building out key features involved building extra infrastructure in the form of many helper classes. It is easy to get lost in a large code base and so the solution to this was making sure any feature that was needed was completed first and foremost. Just as optimizations are a series of backtests, and WFA is a series of optimizations and backtests, this meant modeling and building out the backtest capability first; followed by ensuring the portability and customization of

backtests methods to suit optimization and then making sure both methods could be used in tandem with WFA.

Another challenge was making sure the design of the framework's components were done properly before the implementation. Due to how software engineering can entail many time delays due to improper modeling of situations when code is being built, it was extremely important to make sure the design of the components were modeled properly and then implemented in the most optimal way to ensure against constant refactoring from new additions.

7.3 Contributions to Algorithmic Trading

Once development is finalized so certain features out of the scope for this thesis are implemented, this framework will be available on github for users to use. My goal is to create more and better software tools for algorithmic traders to use so that their process of developing algorithmic trading models can be accomplished.

7.4 Future Work

Future work entails building out more software tools for validating trading models, creating machine-learning based trading models and further exploring the increasing research in the field of algorithmic trading. I want to take this framework and improve it to provide different ways to create, customize, and validate trading models, especially of different financial instruments. Ideally, this will turn the framework into something I can use as a swiss-army-knife of sorts for developing profitable trading models for different financial instruments such as futures, crypto, bonds, and options.

References

Arakelian, Veni & Bolesta, Karolina & Jerić, Silvija & Liu, Yiting & Osterrieder, Joerg & Pott, Valerio & Schwendner, Peter & Sutiene, Kristina & Weinberg, Abraham. (2024). A discussion paper for possible approaches to building a statistically valid backtesting framework. 10.2139/ssrn.4893677.

Arian, H., Norouzi Mobarekeh, D., & Seco, L. (2024). Backtest overfitting in the machine learning era: A comparison of out-of-sample testing methods in a synthetic controlled environment. *Knowledge-Based Systems*, *305*, 112477. https://doi.org/10.1016/j.knosys.2024.112477

Bailey, D. H., Borwein, J. M., & Lopez de Prado, M. (2016). Stock portfolio design and backtest overfitting. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2739335

Bernhardt, D., & Eckblad, M. (2013, November 22). *Stock Market Crash of 1987*. Federal Reserve History. https://www.federalreservehistory.org/essays/stock-market-crash-of-1987

Bessembinder, Hendrik. (2003). Selection Biases and Cross-Market Trading Cost Comparisons.

Bhowmik, R., & Wang, S. (2020). Stock market volatility and return analysis: A systematic literature review. *Entropy*, *22*(5), 522. https://doi.org/10.3390/e22050522

Chojnacki, K., & Ślepaczuk, R. (2023). Ensembled lstm with walk forward optimization in algorithmic trading. *EconBiz*, *2023,15*(422). https://doi.org/10.2139/ssrn.4516294

Gratton, P. (2024, November 21). *Stock Market Crash of 2008*. Investopedia. https://www.investopedia.com/articles/economics/09/subprime-market-2008.asp

Hayes, A. (2024, May 31). *Position Definition—Short and Long Positions in Financial Markets*. Investopedia. https://www.investopedia.com/terms/p/position.asp

Hayes, A. (2024, August 13). *What Is a Trading Strategy? How to Develop One*. Investopedia. https://www.investopedia.com/terms/t/trading-strategy.asp

Hill, A. (2019). Finding consistent trends with strong momentum - RSI for trend-following and momentum strategies. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3412429

The History of NYSE. (2024). Www.nyse.com. https://www.nyse.com/history-of-nyse

Lawrence Damilare Oyeniyi, Chinonye Esther Ugochukwu, & Noluthando Zamanjomane Mhlongo. (2024). Analyzing the impact of algorithmic trading on stock market behavior: A comprehensive review. *World Journal of Advanced Engineering Technology and Sciences*, *11*(2), 437-453. https://doi.org/10.30574/wjaets.2024.11.2.0136

Lo, A. W., & MacKinlay, A. C. (1990). Data-Snooping Biases in Tests of Financial Asset Pricing Models. *The Review of Financial Studies*, *3*(3), 431–467. http://www.jstor.org/stable/2962077

Mazur, M., Dang, M., & Vega, M. (2021). COVID-19 and the march 2020 stock market crash. evidence from s&p1500. *Finance Research Letters*, *38*, 101690. https://doi.org/10.1016/j.frl.2020.101690

Qin, Xiongpai & Wang, Huiju & Li, Furong & Chen, Jidong & Zhou, Xuan & Du, Xiaoyong & Wang, Shan. (2012). Optimizing parameters of algorithm trading strategies using MapReduce. Proceedings - 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2012. 10.1109/FSKD.2012.6233799.

Reiff, N. (2025, March 31). *Stock Order Types Explained: Market vs. Limit Order*. Investopedia. https://www.investopedia.com/investing/basics-trading-stock-know-your-orders/

Richardson, G., Komai, A., Gou, M., & Park, D. (2013, November 22). *Stock Market Crash of 1929*. Federal Reserve History. https://www.federalreservehistory.org/essays/stock-market-crash-of-1929

Xue, R.-B., Ye, X.-S., & Cao, X.-R. (2021). Optimization of stock trading with additional information by limit order book. *Automatica*, *127*, 109507. https://doi.org/10.1016/j.automatica.2021.109507

Unger, A. (n.d.). How to Use Walk Forward Analysis: You May Be Doing It Wrong! Ungeracademy. https://ungeracademy.com/posts/how-to-use-walk-forward-analysis-you-may-be-doing-it-wrong