

Tri Nam Do

## **DEVELOPING AN ALGORITHMIC TRADING BOT**

## **Developing an Algorithmic Trading Bot**

Tri Nam Do  
Bachelor's Thesis  
Spring 2021  
Bachelor's Degree in Software Engineering  
Oulu University of Applied Sciences

## ABSTRACT

Oulu University of Applied Sciences  
Bachelor's degree in Software Engineering

---

Author: Tri Nam Do

Title of the thesis: Developing an algorithmic trading bot

Thesis examiner(s): Teemu Korpela

Term and year of thesis completion: Spring 2021

Pages: 49

---

The idea of cloud computing was conceived a few decades ago but only until recently has it received considerable notice from the global developer communities, particularly when made remarkably more approachable and affordable for the SMEs and recreational developers with AWS Free Tier and Azure free accounts. However, the big picture of how all components of a cloud service provider work together is still quite challenging to comprehend. Therefore, the outcome which this thesis aims to achieve is to demonstrate how the most popular services provided by AWS can cooperate all together in an exemplary cloud application – an automated stock trading bot.

This thesis also provides a descriptive explanation on the fundamentals of stock investing and trading as well as some popular trading strategies, including ones with the aid of Machine Learning models to predict the price movement of certain stocks so that readers without a background in Investing or Data Science can follow through the reasoning of how the program was designed in a specific way. It is widely believed that the stock price is implausible to be predicted, so this project did not dive deep into optimizing the results but to present a suggestion as to how the use of Machine Learning can fit in the big picture.

Overall, the result of this project was within expectation, meaning that a functional prototype of the program was produced with the capability of further development and scaling up. This was accomplished by setting up a logical design of the program, gradually making each of them functional and connecting them together so they can function as a whole.

---

Keywords: Trading, AWS, Automation, Serverless

# CONTENTS

CONTENTS .....	4
TERMS .....	6
1 INTRODUCTION .....	7
2 STOCK INVESTING ESSENTIALS .....	8
2.1 Elemental concepts .....	8
2.2 Trading strategy .....	9
2.2.1 Definition .....	9
2.2.2 Development of a trading strategy .....	10
2.2.3 Technical indicator .....	10
2.2.4 Going long and going short .....	11
2.2.5 Trading frequency .....	12
2.2.6 Strategies and technical indicators used in this project .....	12
3 PROGRAM OUTLINE .....	14
3.1 Topological diagram .....	14
3.2 Deployment infrastructure .....	15
3.3 Development environment .....	16
3.3.1 Operating system .....	16
3.3.2 Programming language .....	16
3.3.3 Code editor .....	17
3.3.4 Project encapsulation technology .....	18
3.3.5 Version control system .....	20
3.3.6 Database management tools .....	20
4 DATABASE .....	22
4.1 Functional requirements .....	22
4.2 Non-functional requirements .....	22
4.3 Database schema .....	24
4.4 Development and deployment .....	25
4.4.1 Development phase .....	25
4.4.2 Deployment phase .....	27
5 DECISION-MAKING MODULE .....	28
5.1 Functional requirements .....	28

5.2	Non-functional requirements .....	28
5.3	Development and deployment.....	28
5.3.1	Simple Moving Average Crossover .....	29
5.3.2	Machine Learning for classification.....	31
5.3.3	Deployment phase .....	38
6	ORCHESTRATION MODULE .....	39
7	BUSINESS LOGIC MODULE .....	40
7.1	Functional requirements.....	40
7.2	Non-functional requirements .....	40
7.3	Development and deployment.....	41
7.3.1	Database interaction methods .....	41
7.3.2	Data preparation methods .....	41
7.3.3	Program data and configuration loading .....	42
7.3.4	Model classes .....	43
7.3.5	Deployment phase .....	43
7.3.6	Results.....	46
8	CONCLUSION .....	49
	REFERENCES .....	51

## TERMS

API	Application Programming Interface
DB	Database
ER	Entity Relationship
IDE	Integrated Development Environment
ML	Machine Learning
OHLC	Market data which includes Open, High, Low and Close price of a stock

# 1 INTRODUCTION

In 2011 McKinsey & Company, which is a consulting company in the field of Management and Administration, published a report stating that exploiting the power of big data will become the main competing edge for existing companies and new competitors with data-driven operations will be introduced to the market. Indeed, after the computerized trading systems were put into use in American financial markets in the 1970s, there has been an increasing number of algorithmic traders, which is a term referring to market traders who use algorithms to build up their trading strategies. This is facilitated by the computation capacity of computers nowadays, which allows for the handling of tens of thousands of transactions per second.

However, for novice algorithmic traders, it is also important that the expenditure of deploying and maintaining their bots must be minimal, which means the bot should use as little resource as possible and only allocate more when needed. "Big things start small" – once said Jeff Bezos, the richest American in 2020 [2], which is why this document aims to describe a simple design of a scalable algorithmic trading bot first, which can be run on even one's personal computer based on an explanation of the fundamental knowledge of trading as a foundation.

The learning outcome this project desires to achieve is to have a fundamental understanding of cloud computing and to explore the software ecosystem provided by Amazon under the name Amazon Web Services (AWS) and how the collaboration of its components can aid human in one of the most demanding decision-making tasks: stock trading. This was out of personal interest as well as to have better preparation for a future career in Data Engineering.

This project employs two methods to make trading decisions: using existing trading indicators as well as using a Machine Learning model to predict the price and acting accordingly. The aim of this project is not to create a top-of-the-line and lucrative trading bot but a working prototype of such a program to lay the foundation to enter the field of Data Engineering and Cloud Computing because the code in this project is designed to be compatible with serverless computing and AWS services particularly.

## **2 STOCK INVESTING ESSENTIALS**

There is a massive amount of information and rules one has to learn when it comes to trading, especially if he is self-employed and has no desire to rely on any consultancy from third-party agencies. Therefore, within the scope of a thesis, only the core concepts should be covered, and it is worth remembering that the definitions and descriptions provided in this document are under the financial market context.

### **2.1 Elemental concepts**

The most indispensable component of a market is, needless to say, the goods or services which hold a certain monetary value and can be exchanged within the market. In financial markets, they are referred to as financial instruments [3] and can be categorized into three types: equity, debt, and one which is a combination of the two named hybrid [3]. A financial instrument in a stock market is typically referred to under the term "security", which indicates the partial ownership of the stakeholders over a joint-stock company or a governmental entity [4].

The action of exchanging these securities and receiving back money or any other type of asset as compensation is called trading, which can take place between sellers and buyers [5]. In a stock market, a trader can buy more securities or sell ones in his possession, so he acts as a buyer and a seller.

To ensure the fair and well-organized trading of securities as well as improve the efficiency of announcing their price, there are platforms, called exchanges, have been established for governments, companies and organizations to sell their stocks to investors [6].

Since the size of an exchange is limited, traders who are not a member of it must go through individuals or firms who are part of the exchange. They are called brokers [7], and they can get benefits by accepting commission or service fees paid by outside investors or by the exchange itself. Since those investors cannot trade securities directly by themselves, they must give a set of instructions, which is called an order, to these brokers to carry out on behalf of them. The execution



of an order results in a position in the market, which refers to the amount of a security owned by that trader.

Combined all the terms above, one can visualize a financial market at its simplest, as demonstrated in figure 1 below. A security exchange is established so that its participants can buy or sell stocks, which are published by companies and organizations. They can be investors or brokers, who act as an intermediary and receives orders to buy or sell from third-party traders. However, understanding the core concept is one thing and becoming a successful trader is another, which takes much more effort and years of real-world experience to achieve.

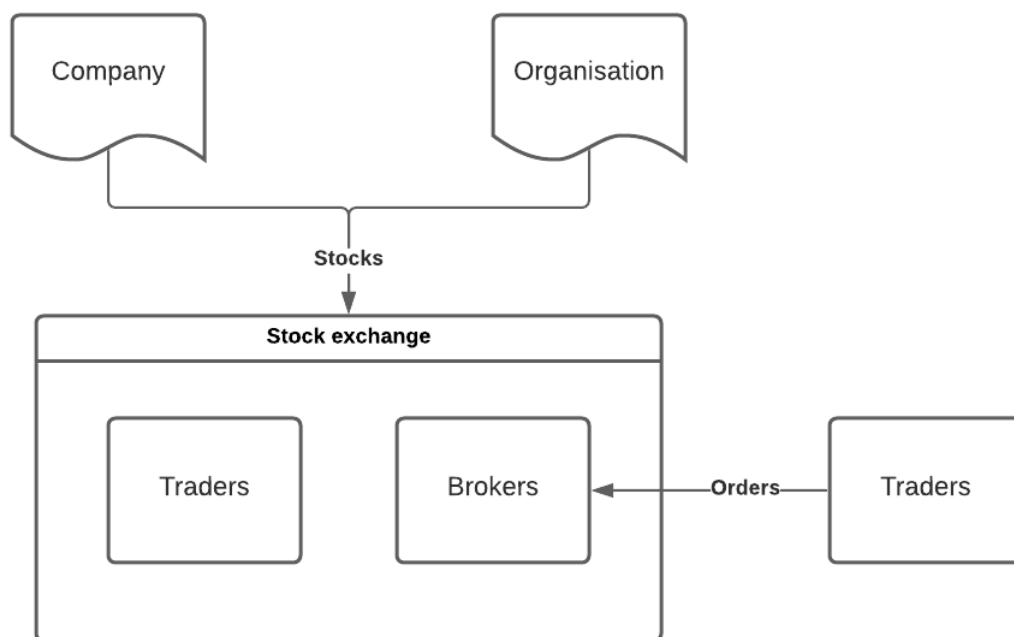


FIGURE 1. Simplified illustration of a stock exchange system

## 2.2 Trading strategy

### 2.2.1 Definition

A trading strategy is a set of predefined rules, which is applied to make decisions regarding the security to buy or sell [8]. It typically includes a plan with specific investing goals, risk management, time horizon and tax implications. Generally, there are three stages in a strategy, each of which is entailed with metrics of measurement to assess and modify the plan according to market changes. These stages comprise planning, placing trades and executing them [8]. Firstly, one must evaluate

the exchange status and based on that, combining his predefined rules, determine the selling and buying decisions. Subsequently, these decisions are sent to a broker to execute them. To access the effectiveness of a strategy, it can be back-tested, or in other words, put in a simulation environment with historical market data for its performance to be measured.

### **2.2.2 Development of a trading strategy**

Essentially, there are two ways of developing a strategy, which are to make use of technical or fundamental data [8]. Traders using the former kind typically observe the price of a security and its movements to determine whether a security is worth buying and how much should be bought. While, the latter, as the name suggests, takes into account basic quantitative and qualitative data such as the financial status and the reputation of a company to assess its security's stability and health. One might argue that quantitative trading can be considered the third type [8], but it is similar to technical trading in the sense that it makes use of numerical data but of a considerably larger complexity with many data points in use.

### **2.2.3 Technical indicator**

Developing a trading strategy from a technical approach relies heavily on observation of a set of data signals called technical indicators. They are produced by aggregating security data using certain mathematical formulas so that it becomes much quicker and more comprehensive for one to make out the trend and price movement of a security from a quick look to cope with the fast pace changing of market price [9]. Some of the most common ones include Relative Strength Index (RSI) and Moving Average (MA). Traders typically establish their selection of indicators depending on the goal and the strategies being used.

The experiments carried out during this project make use of the MA indicator. How one can calculate this is that on any trading day, he first defines a specific period of time to retrieve historical data, for example, 20 days, then calculates the average closing price of that security during the previous 19 days and the chosen one and keeps doing this recursively until the present. Looping over a longer period, this results in a smoother line, helping investors make out the overall trend, unaffected by the daily volatility of the stock price (figure 2).



FIGURE 2. Demonstration of how Simple Moving Average helps to detect the overall trend [10]

## 2.2.4 Going long and going short

The goal of any sensible trading strategy is to make a profit. The approach which is most widely known, even by novice traders, is going long. It means the investors buy a certain number of shares at a price, then hope to sell them at a higher price in the future, thus making the profit by the increase of the price per share. While that seems of common sense, there is a number of investors, usually professional ones with experience, would do the exact opposite, which is to sell stocks at a high price and wait for their price to drop. This strategy is called to go short. The mechanic behind how this works is quite creative: the investor following this strategy will borrow a number of shares from a broker or other investors and sell them at a higher price. At this point, this investor will have an amount of money profit from selling the stocks and debt of that number of stocks. When the price of those stock drops, he will buy them back, this time at a lower price and returned the number of shares he borrowed. This way, the investor makes a profit from the price difference between selling and buying.

This introduces the concept of take-profit and stop-loss orders. In general, they can be understood as upper and lower thresholds of the price of a stock for an order to be sold, which are defined by individual traders. A position will be closed, or in other words, all shares in that order will be sold, if the price of that stock reaches the take-profit value. This is for shareholders to prevent holding a position for too long and bypass the highest price in that period [11]. The mechanic of a stop-loss

order is reversed of what was described above: the position will be closed when the stock price falls to the lower threshold to mitigate loss [12]. However, these thresholding values are used in a more versatile manner in real-world situations. For example, when an investor is in a short position, the take-profit value can be used as a discipline for him to know that it is time he bought back those shares he borrowed before their price goes even higher and he ends up with a more considerable loss.

### **2.2.5 Trading frequency**

This is the term referring to the number of orders being placed by a trader in a set interval of time. This can range from every fortnight, every day to as high as every 0.003 seconds. Experimented in this project are intraday trading, or day trading, which is to act upon the price changes every second and make profit out of very small price changes; and position trading, which is to check the stock price daily to decide if the position should be held or closed, thus an order in this kind of trading can last for months or years.

The plan for this project is to experiment with strategies belonging to both ends of the spectrum, which means the program can be executed once every day just a short while before the market closing time to trade, then wait for the stock price to go up or down for several days. It can also be kept running for as long as the market is open, checking the price every second and attempting to compute the future price to decide whether to buy or sell. This method of trading is called scalping.

### **2.2.6 Strategies and technical indicators used in this project**

There will be two approaches taken in this project and both of them will be backtested, in other words, testing the trading strategy against historical market data, using the same Python package. The first one will make use of the most common technical indicators being used by traders, including the simple moving average indicators. For this, data in an appropriate format must be prepared, a function to compute the said indicators must be written and the package mentioned above will be used for testing the strategies devised based on those indicators. The second approach involves simplifying the prediction task into forecasting whether the price of a model goes up or down in the next two days, which turns this into a classification problem. Several Machine Learning models will then be trained and compared with one another to select the model with the

best performance. Needless to say, the selected one will be used for testing with the same Python module mentioned above, in which the strategy will be going long when the price is predicted to go up and going short if it is anticipated to decrease. The model training task is iterative, which means when a certain amount of new data is added, the models will be retrained so that they can keep up with the latest market trend.

### **Moving Average (MA) and Simple Moving Average Crossover**

This strategy relies solely on two Moving Average indicators, one over a shorter period of time and one over longer. The way a trader interprets the mechanic of this strategy is that when the shorter-term MA crosses and overtakes the longer-term one, it means that the market trend is most likely to go up, which is the signal for investors who are going long to buy shares or keep holding their long positions. A reverse pattern applies when the shorter-term drops below the one on a longer-term, which is when the price of that stock has a likelihood to go down.

### **Bollinger Bands**

Just as the moving average indicator, this is also a lagging one, which means it only demonstrates and confirms market trends that happen in the past. It consists of two lines, which are formed by connecting the points one negative and one positive standard deviation away from the average line of a 20-day roll by default [13]. Depending on the use case, this distance can be defined by traders themselves to fit best with their strategies.

Since these lines are computed based on the standard deviation, which represents how varied the values are, or in this case, how volatile the stock price is, the width of a Bollinger band can be an indicator of the market volatility. When the band is narrow, it means that the price does not fluctuate considerably and the market is consistent. This is typically followed by a period of volatility, during which traders can find more investing opportunities [13].

The band also gives information about the status of the market, whether it is oversold or overbought, which means the stock is being sold at a price lower than its actual value [14] or at a higher one respectively. Therefore, this can also be used in fundamental analysis to evaluate an asset before actual purchases.

Much as popular and useful this may seem to novice traders, it tends to be mistaken for a signal to trade, which is utterly not the case [13]. For example, when a stock experiences a boom in its price and exceeds the upper bound, though a major event, it in no way implies that this stock is profitable and should be bought immediately. Therefore, John Bollinger – inventor of this indicator - recommended that it should be used in combination with other information when assessing the market [13], which is why its concept is introduced here and later will be used for enriching the training dataset for the Machine Learning models to predict future market trend.

## 3 PROGRAM OUTLINE

### 3.1 Topological diagram

With the help of an online tool named LucidChart, a simplified diagram can be constructed to demonstrate the building blocks of this trading bot:

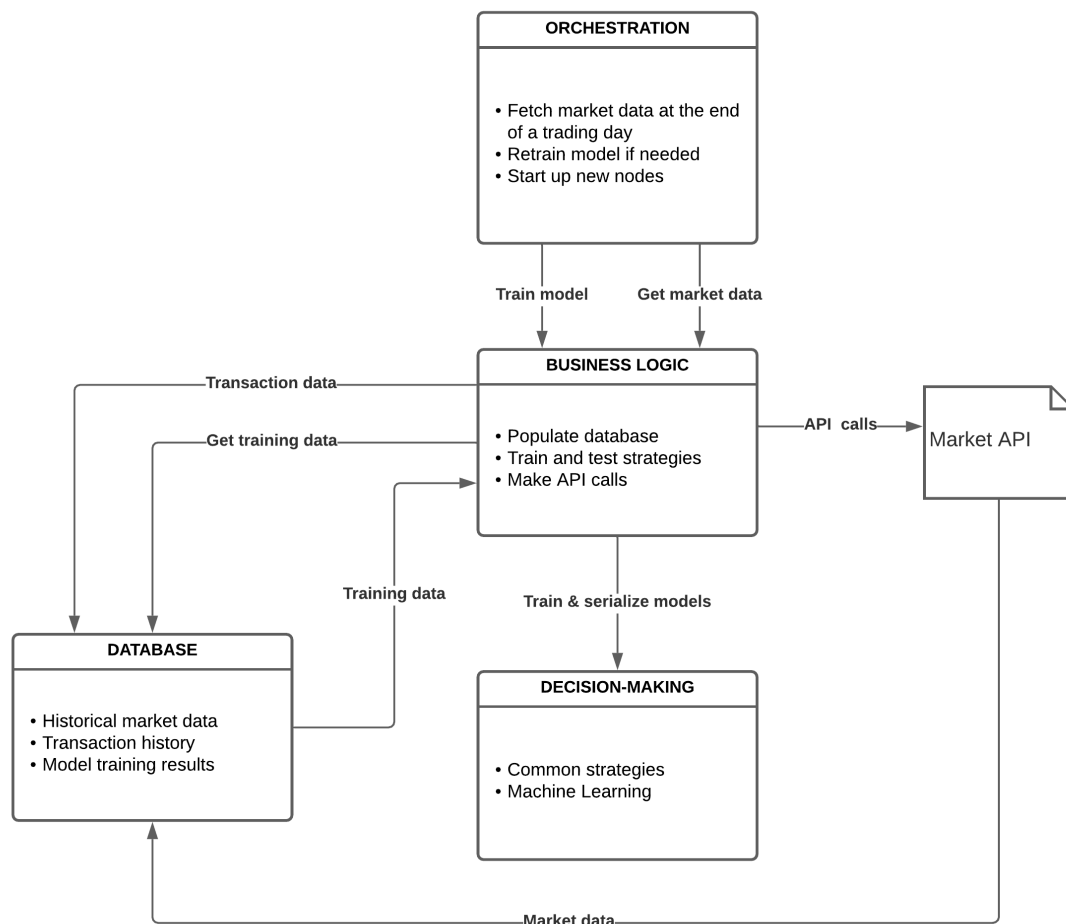


FIGURE 3. Main interactions among components of this program

The structure was designed so that each component takes care of a specific task, which turned out to be advantageous because the code and the project progress became much more manageable. In this way, a requirement for the whole program can be translated into a group of specific tasks with ease. A good demonstration of this is that from the requirement that the bot must be capable of position trading, one can interpret that this typically involves computing an indicator whether to buy or sell and invoking that computation task daily at a specific moment. Therefore, the code

needed to be changed or written should reside in the orchestration and the decision-making modules, and this example, it includes a function to compute the trading decision and a *cron* expression to schedule the invocation of said function.

After a glance at the diagram (figure 3), one should be able to tell that the Business Logic module is the backbone of this program. It invokes methods from other modules to perform tasks such as inserting data into the database or training and evaluating Machine Learning models from the decision-making module. Since each module was designed for one distinct purpose, the codebase can easily be added with more modules as listed in the suggestion section later. A more descriptive explanation for each component will be described in the subsequent sections.

### 3.2 Deployment infrastructure

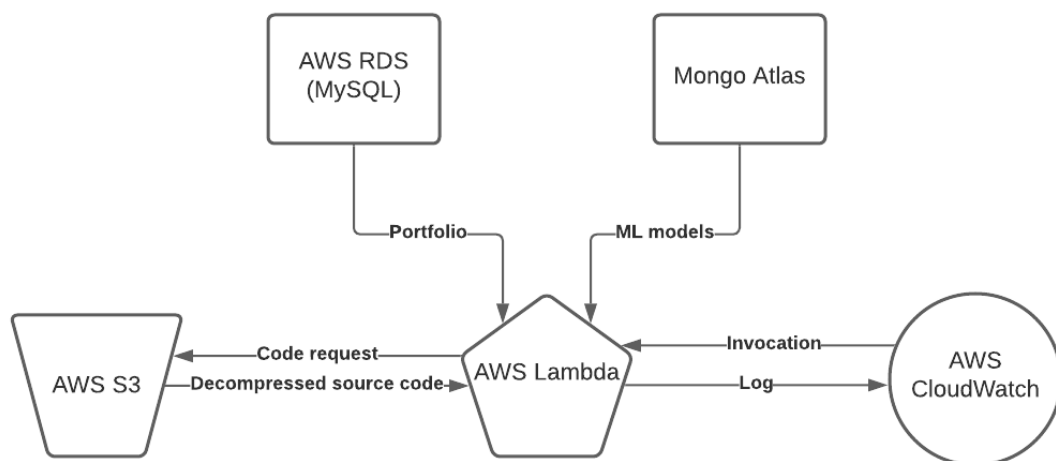


FIGURE 3. Interactions between cloud services for position trading

The data persistence solution of choice was to use the MySQL database service provided by AWS for tabular data such as market price and historical transactions, while the Mongo Atlas is the deployment platform for MongoDB, which is capable of storing non-tabular data such as Machine Learning models. A thorough reasoning and implementation description of this can be seen in chapter 4.

This project took advantage of the AWS Free Tier services granted to newly created accounts. Since the computing capacity is quite limited on this tier, the services should only be used for

position trading as it does not require substantial resources on a small scale and does not have to cope with real-time processing, which is inherent to day trading strategies.

Using the Serverless framework, a majority of infrastructure provisioning and maintenance is executed automatically. Overall, the source code and its dependencies were packaged, then zipped and uploaded to an AWS S3 bucket using the Serverless framework. An AWS CloudWatch rule was then created, also by the Serverless framework based on predefined configurations of the developer, to invoke the AWS Lambda function just a short while before the market closes. Upon activation, the Lambda function first loads the source code and its dependencies from the designated S3 bucket, then executes code to fetch the trading profile stored in the databases as well as the market data on that day and calculates the subsequent actions based on said data. If there are any new transactions, they will be written to the log file as well as sent to the database to be persisted for possible future analysis. A deeper explanation of this deployment infrastructure will be provided in the following chapters.

### **3.3 Development environment**

#### **3.3.1 Operating system**

The host system is Windows 10, with an Ubuntu version 18.04 installed on the Windows Subsystem for Linux. All the Docker containers used in this project are Linux-based, which can be interacted with using a *bash* shell initiated by the command

```
docker exec
```

However, having Linux as the main system is much recommended, as Linux provides a faster way to install Python packages, especially with those in C and C++ which require to be compiled first such as *pystan* and *Facebook Prophet*.

#### **3.3.2 Programming language**

The programming language having been used in this project was Python. This was because of the following reasons:



- Large community support: Python is a multi-purpose programming language, but thanks to its minimalist syntax, it is commonly used for data processing and automation, which is also the theme of this project. Therefore, it was much easier to get help from others if errors happened.
- Simplicity: the syntax of Python is outstandingly minimalist. It is not strongly-typed and does not involve an excessive number of classes, which streamlines tasks such as data type conversion.
- Third-party libraries: Python programming language has numerous packages for handling data processing, and they can help to abstract the code even more while keeping it clean. In this way, the amount of actually written code can be reduced while its readability increases.

### **3.3.3 Code editor**

A code editor named Visual Studio Code (VS Code) was chosen as the main and only text editor for this project. Although it might not provide as many tools as a discrete IDE such as PyCharm, there are numerous advantages it is in possession of:

- Small footprint: the software takes up little space compared to PyCharm, which requires the installation of Java for it to operate smoothly. VS Code also takes much less time to start up, even with certain plug-ins enabled for Python development.
- Interpreter auto-selection: to ensure consistency in dependency versioning, a Python virtual environment is used. When this is placed in the root directory of the project, VS Code can automatically detect it and use it as the interpreter to execute the code. Even the terminal window will also switch to the virtual interpreter automatically upon activated.
- Developer's familiarity: This is downright important because it has a direct effect on the workflow and the productivity of a developer. Under the circumstances of this project where its progress must be brisk, VS Code is the way to go because the author has the most experience with Visual Studio IDE and Visual Studio Code.

VS Code can also be added with several useful extensions while keeping it lightweight and fast.

### 3.3.4 Project encapsulation technology

To isolate the source code of this project from other system processes to mimic as close to the deployment environment as possible, these technologies were used:

#### Python virtual environment

The data processing code in Python typically uses several packages which do not come as built-in ones. However, they are also constantly improved and updated, during which certain API functions are deprecated. Therefore, to make sure that the code can always be executed without errors, the version of such packages must be kept unchanged. A discrete installation of Python may not be compatible with every script [15], which is why each Python program should be run in its own environment.

The Python virtual environment helps to achieve this by creating a fresh installation of Python in a local directory, initially with no third-party packages. Those added during the development phase using a package manager of this local environment are kept in the same directory and separated from the global environment installed for the whole machine. In this way, using the local interpreter, the program will always use the environment set up specifically to meet its requirements, and thus, is unaffected by future updates from the package providers.

In this project, a Python virtual environment was initiated by running the following command inside the project root directory, which created a subdirectory named *env* and stored the simulated Python interpreter as well as its relevant scripts in that folder:

```
python -m venv ./env
```

Third-party packages added in the future are also kept here to be isolated from the global Python installation. A list of the required libraries and their version numbers can be stored in a *.txt* file using the command:

```
pip freeze > requirements.txt
```

which can then be used in the command:

```
pip install -r <path to .txt file>
```

to install necessary packages in a new Python environment or roll back to the previous version if needed.

### Docker container

While the Python virtual environment described above provides a simulated code interpreter, this containerization technology provides an encapsulated executing environment. It does so by grouping certain system processes needed to execute the source code into namespaces, which provide a layer of encapsulation [16]. The local clusters of SingleStoreDB and MongoDB databases used for testing code were initialized in Docker containers, which have the following advantages:

- No need for actual installation: Docker images, or in other words, the blueprints mimicking the actual systems of these database systems were pulled from a centralized platform called Docker Hub, then modified by a few lines of code to tailor them to the requirements of this project and they were ready to serve.

```
def start_db():
    setup_env_variables()
    subprocess.run('docker-compose up')

def setup_env_variables():
    license_key, root_pwd, _ = get_db_configs()
    os.environ['LICENSE_KEY'] = license_key
    os.environ['ROOT_PWD'] = root_pwd
```

Figure 4. Python code for starting database containers.

- Faster setup: With SingleStoreDB and MongoDB databases listed as services in a *docker compose* file, all it took to get their server up and running was one command to execute a Python script (figure 4), which is highly convenient. However, it is worth keeping in mind that Docker relies on the Windows Subsystem for Linux to operate normally, which can occupy as much as 10GB of RAM.

Since the development of this project is a Windows machine, a feature on Windows called Windows Subsystem for Linux must be enabled so that Docker can run smoothly.

### 3.3.5 Version control system

To keep things under control, the source code of this project must be organized and managed sufficiently to ensure that the project progress is unaffected by messy code. A version control system (VCS) also allows developers to roll back to the previous state of the source code if the current one turns out to contain multiple bugs or there is an urgent task emerging which makes the currently developed feature of lower priority.

The VCS selected for use in this project was Git, mostly because it is free of charge and familiar to the author. A remote repository was created on GitHub to make sure that there was always a clean and working version of the app persisted in a remote repository.

### 3.3.6 Database management tools

The SingleStoreDB Docker image is shipped with a built-in management studio which can be accessed via a web browser by sending HTTP requests. It includes several monitoring features such as a dashboard, visualization, database node cluster health and a text editor for the user to execute SQL queries. Though seems adequate, it is not a very convenient tool to work with, which is understandable because it is basically a web app and does not support many keyboard shortcuts. In this case, its most frequently used feature was the SQL Editor for checking if the content of the database had been changed after new Python functions were written to interact with the database tables.

The Docker image of MongoDB was delivered with a command-line interface (CLI), which can be initiated using the command:

```
docker exec -it <name of the MongoDB container> mongo
```

The Windows *PATH* environment variable must be set to point to the Docker binary for this command to work. After executed, it opens a database session in the same command prompt windows where users can interact with the database by writing Mongo queries.

An interesting discovery in the later stage of this project was MySQL Shell, a program which provides a CLI for interacting with SQL database system using JavaScript, Python or SQL code.

The compelling feature of this program is that user can easily switch among these modes within a database session by typing the command `\js`, `\py` or `\sql` respectively. This was used as a MySQL client to connect to the AWS Relational Database System in the deployment phase for testing.

## 4 DATABASE

This module is responsible for storing the trading profile, market data, activity logs of the bot as well as the Machine Learning models used to make a decision. Due to its vitality, specific requirements must be devised well before its actual implementation to keep the development phase on track.

### 4.1 Functional requirements

The most important requirement for a database system is data persistence. All entries must be kept securely, complying to predefined format and structured, or schemas, which will be described later in the Database schema section.

Users must be able to log into the database with their credentials and make modifications if needed. However, the database system must be able to prevent them from accessing certain parts without authorization.

The program must also be able to store and retrieve its trading profile and trained ML models for future usage and comparison from designated databases.

### 4.2 Non-functional requirements

In order to establish these, use cases and an estimation of the actual workload must be examined. From the proposed design described in section 3.1 about the program topology, it can be concluded that this database system will persist trading profile information, market data, transaction history which the bot has made and the results of Machine Learning model training if available. Since this type of data is tabular, meaning that the number of attributes of stocks remains the same, a relational database management system (RDBMS) should be put into use. Since its introduction in 1970, there has been a proliferation in the number of systems using this technology, each with its own strengths and weaknesses. However, there is no one-fit-all contender but a trade-off between the performance, occupied space and cost of deployment and maintenance.

Data integrity must also be remained over time and changes, which is to make sure that data format and types are correct, foreign key columns do not refer to non-existing records or tables or any types of discrepancy which can cause the program to crash due to data type error or null reference error.

In this project, the goal is not to have a profitable trading model at the production level but a working prototype of such a program, which is preferred to be scalable as well. This trading program was also not set out to run day trading strategies, which requires the capability of real-time processing. Therefore, it was not of high priority that the reading and writing operations were particularly fast. For the sake of experimentality, the database system should also be compatible with other software development tools such as Docker containers and can be interacted with using Python, which was chosen to be the main language of this project. Additionally, the database should be free of charge to use as well.

The article written by P. Sanghvi [17] stood out among several online ones which were consulted, exalting the efficiency and reliability of SingleStoreDB and ClickHouse database. The ClickHouse database was then found to be evaluated in another article published by Percona [18], which helps narrowing down the database of choice to SingleStoreDB. Just like any other database systems which can be managed using Structured Query Language (SQL), SingleStoreDB can be interacted with using a Python module named MySQLdb. In this document, such kind of databases will be referred to as SQL databases.

For persisting ML models, the chosen database technology is MongoDB, mostly because of its support for non-tabular data such as those models and the author's familiarity in using it. A Python package named *pymongo* is used for interacting with this database.

### 4.3 Database schema

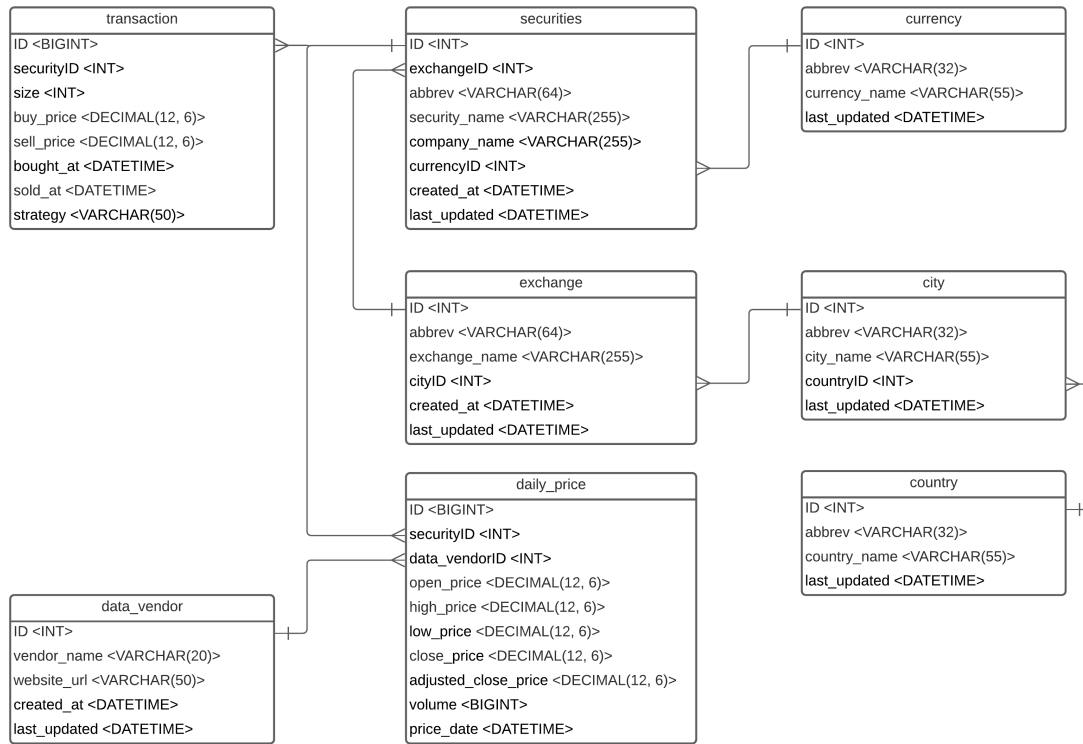


FIGURE 5. ER diagram of database tables in SingleStoreDB.

The diagram above is relatively self-explanatory because it was designed to mimic as close to the actual example as possible while keeping things simplified. The *daily\_price* table might seem redundant because such data can be fetched from virtually anywhere. However, if one were to improve ML model performance, data such as the measurement of *beta* or *P/E* might be pulled from different sources so there is a need for a central data storage, but it is not highly relevant at the moment of development because this thesis focuses on trading strategies based on technical analysis. It can also be a backup dataset for training new models in case historical market data cannot be fetched from the original source due to server downtime or connection errors.

The current trading profile is stored in the table *metadata*, which has the following information:

- *balance*: the amount of money uninvested in any stocks. By default, this was set to be 4000.
- *currencyID*: an integer value referring to a row in the *currency* table. Initially, the currency was USD.



- *current\_strategy*: a string value as the name of the currently employed trading strategy. There are two groups: those which start with “ML\_” and those do without. As one might have guessed, the former one indicates that a Machine Learning model is being used, thus the script will invoke a function to fetch it from the MongoDB database. The latter one refers to strategies which rely on technical indicators, so the main script will just load the prebuilt class accordingly.

Beside this, a list of instances of the model class *Stock* (which, together with model class *Portfolio*, will be discussed in chapter 7) was binarized and stored in a collection named *owned\_stocks* in the MongoDB.

For storing ML models, there are no rigid schemas which have to be followed, so a collection named *models* was created in a Mongo database named *algotrading\_models*, where documents were stored with at least the following fields:

- *model*: the actual ML object in Python which was “pickled”, or in other words, turned into a byte array and saved to the database as a string.
- *model\_name*: the name of the model above, which was ensured to be unique for easier retrieval.
- *created\_at* & *last\_trained*: the time when the model was created and trained respectively.

## 4.4 Development and deployment

### 4.4.1 Development phase

In the development phase, the two databases were encapsulated in Docker containers. Essentially, this is a technology which allows the installation of software in an isolated environment to mimic as closely the actual situations as possible without the need to set up Virtual Machines or buying actual machines to run the needed services.

Overall, the documentation of both database technologies in use as well as their Python connectors was straightforward, which helped to smoothen out the learning curve so there were few obstacles encountered when working with this.

Upon selecting the database technologies, one should include in the source code methods to automatically set up and interact with such systems. In this project, database configurations such as license key, root username and password as well as database name and listening port assignment were kept in a file named *df\_config.ini* and parsed using a Python package called *configparser*. User credentials were parsed from the configuration file prior to any interaction with the database.

The database schema demonstrated above was also kept in this module, as strings of *CREATE TABLE* queries (figure 5). The same went with the queries for inserting and deleting entries.

```
#region CREATE TABLES
def create_exchange_table():
    return f'''
        CREATE TABLE IF NOT EXISTS {exchange_table_name}
        (
            id INT AUTO_INCREMENT,
            abbrev VARCHAR(64),
            exchange_name VARCHAR(255) NOT NULL,
            cityID INT NULL,
            created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP(),
            last_updated DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP() ON UPDATE CURRENT_TIMESTAMP(),
            PRIMARY KEY (id)
        ) AUTO_INCREMENT = 1;
    ...

def create_securities_table():
    return f'''
        CREATE TABLE IF NOT EXISTS {securities_table_name}
        (
            id INT AUTO_INCREMENT,
            exchangeID BIGINT NOT NULL,
            abbrev VARCHAR(32) NOT NULL,
            security_name VARCHAR(255) NOT NULL,
            company_name VARCHAR(255) NOT NULL,
            currencyID INT NOT NULL DEFAULT 1,
            created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP(),
            last_updated DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP() ON UPDATE CURRENT_TIMESTAMP(),
            PRIMARY KEY (id)
        ) AUTO_INCREMENT = 1 ;
    ...
```

FIGURE 6. Example table schema queries

These functions were grouped into a list and were executed when setting up the database to create predefined tables and a superuser to manage the database. As part of the setup script, default values such as initial balance and currency were also inserted from the beginning to ensure that a valid trading profile was created.

With regards to setting up the Mongo database, it was fairly simple in the development phase. Since the database system is wrapped in a local Docker image, the only thing to do was to write a

function which reads the host and port data from the configuration file and connect to the Mongo cluster inside that image.

#### 4.4.2 Deployment phase

The SingleStore database requires significantly more effort to be properly deployed to AWS. Considering the scale of this project, it was decided that the service for deployment was AWS Relational Database Service (RDS), and the database technology in use was MySQL because the existing queries were already in SQL and it was also one of the default options provided by Amazon. This helps to level out the learning curve as well as avoiding wasting time on refactoring code. However, there were still quite a few arrangements to be done when setting up the AWS environment, especially when this project used a brand-new AWS account to take advantage of the Free Tier offering.

It is downright a good practice to disable public access to the database instance to prevent it from being exploited by unauthorized access, which accumulates to an unnecessarily high monthly bill. After this, the instance should be attached to a Virtual Private Cloud (VPC) so that it can interact with other AWS services which are used in this project such as AWS Lambda. This was a required step when setting up a new RDS instance.

To ensure easier debugging of deployment, each component was deployed to the cloud platform gradually and individually. For example, one should deploy the database and test to see if the code in the local environment works well with it first before uploading such code to the cloud. In this way, if an error occurs, narrowing down and tracing back to its source is an effortless task. However, since the local computer which hosts the development environment is not a part of the VPC, a security group which allows traffic from and to the IP address of the local machine was also assigned to the database instance.

A database cluster hosted on Mongo Atlas can be logged in using a connection string with the *mongo+srv* protocol, which requires a package named *dnspython* to interact with the cluster. It is also vital to add the IP address of the local machine to a list of the inbound traffic of this cluster so that code running on the local machine can be tested against this newly initialized Mongo cluster before being deployed to AWS Lambda.

## 5 DECISION-MAKING MODULE

### 5.1 Functional requirements

This module is determined to be responsible for making trading decisions, which includes whether to buy or sell as well as the number of shares involved. Therefore, given a set of market data which consists of OHLC data and certain technical indicators, it must be able to carry out computation on them to evaluate the situation and produce the above-mentioned value.

### 5.2 Non-functional requirements

The most important criterion when assessing the progress of developing this module is the profitability of the strategy in use. It will be considered successful and acceptable if the strategy earns over 30% of revenue in return or at least break even respectively.

The method used for evaluating a strategy is to see how it performs over historical market data. For this purpose, there are two popular contenders named *backtesting.py* and *Zipline*. After a few days of research and experiments, *backtesting.py* was selected for its open API and simple installation and test setup.

The same requirements apply when using scalping strategy, but the testing and result evaluation methods are insignificantly different. Because the *backtesting.py* framework does not support intraday trading, a simulated test which loops through each data points must be devised instead of using ready-built tools. In essence, it loops through each row of the market data table, invokes the decision-making function of the bot and calculates the amount of balance left. The scalping strategy is considered successful even if it makes marginal profit or experiences no loss.

### 5.3 Development and deployment

As mentioned in sub-section 2.2.5, this project attempts to make a profit from two approaches: placing orders based on market evaluation using a combination of certain technical indicators or based on the prediction of a Machine Learning model.

To ensure equality in strategy comparison, it is vital that the same testing conditions are applied when testing them individually. Therefore, the initial investment capital was set to be 4000 USD, the commission amount was 2% and the time period over which strategies were tested is from April 2018 to March 2021, or approximately 3 years.

### 5.3.1 Simple Moving Average Crossover

The first strategy to be tested in the experiment was Moving Average Crossover. Using a combination of moving average values over 10 and 15 days, the result turned out to be quite impressive, as shown in the picture below:

```

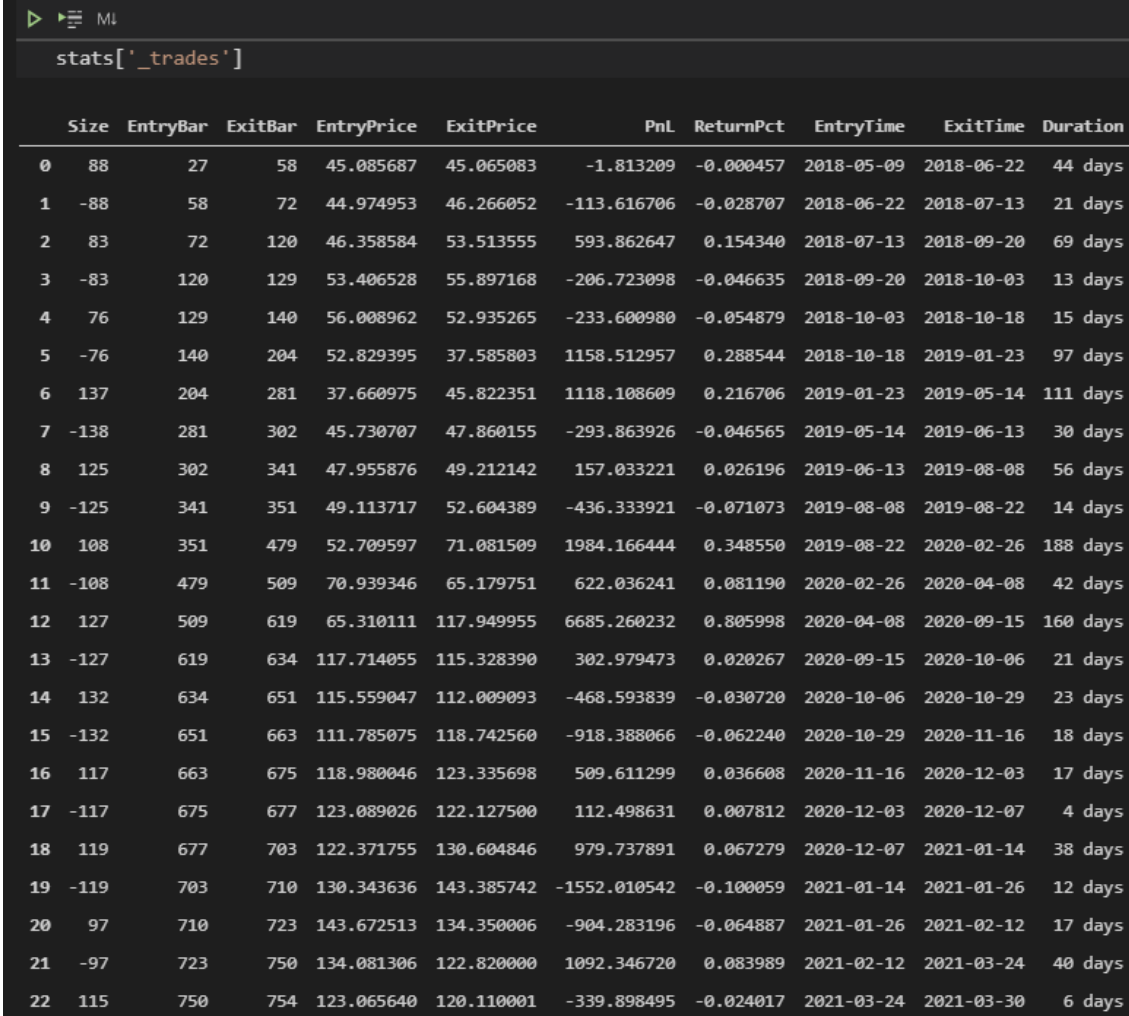
bt = Backtest(data, SmaCross, cash=4000, commission=.002)
stats = bt.run()
stats

Start                2018-04-02 00:00:00
End                  2021-03-30 00:00:00
Duration              1093 days 00:00:00
Exposure Time [%]     96.423841
Equity Final [$]      13847.028388
Equity Peak [$]       17039.512152
Return [%]            246.17571
Buy & Hold Return [%] 198.235923
Return (Ann.) [%]     51.35739
Volatility (Ann.) [%] 50.010715
Sharpe Ratio          1.026928
Sortino Ratio          2.455041
Calmar Ratio          2.085638
Max. Drawdown [%]     -24.624303
Avg. Drawdown [%]     -3.818092
Max. Drawdown Duration 210 days 00:00:00
Avg. Drawdown Duration 20 days 00:00:00
# Trades              23
Win Rate [%]          52.173913
Best Trade [%]         80.59984
Worst Trade [%]        -10.005939
Avg. Trade [%]         5.582115
Max. Trade Duration    188 days 00:00:00
Avg. Trade Duration    46 days 00:00:00
Profit Factor          4.031158
Expectancy [%]         6.987999
SQN                    1.297649
_strategy              SmaCross
_equity_curve           ...
_trades                 Size EntryB...
dtype: object

```

FIGURE 7. Testing result of Moving Average Crossover strategy from Apr. 2018 to Mar. 2021

The amount of money in possession at the end of the testing period was almost 14000 USD, which translates into about 250% of the profit. However, after a deeper inspection of the trade history, there is a few points of observation which should be taken into consideration.



```
stats['_trades']
```

	Size	EntryBar	ExitBar	EntryPrice	ExitPrice	PnL	ReturnPct	EntryTime	ExitTime	Duration
0	88	27	58	45.085687	45.065083	-1.813209	-0.000457	2018-05-09	2018-06-22	44 days
1	-88	58	72	44.974953	46.266052	-113.616706	-0.028707	2018-06-22	2018-07-13	21 days
2	83	72	120	46.358584	53.513555	593.862647	0.154340	2018-07-13	2018-09-20	69 days
3	-83	120	129	53.406528	55.897168	-206.723098	-0.046635	2018-09-20	2018-10-03	13 days
4	76	129	140	56.008962	52.935265	-233.600980	-0.054879	2018-10-03	2018-10-18	15 days
5	-76	140	204	52.829395	37.585803	1158.512957	0.288544	2018-10-18	2019-01-23	97 days
6	137	204	281	37.660975	45.822351	1118.108609	0.216706	2019-01-23	2019-05-14	111 days
7	-138	281	302	45.730707	47.860155	-293.863926	-0.046565	2019-05-14	2019-06-13	30 days
8	125	302	341	47.955876	49.212142	157.033221	0.026196	2019-06-13	2019-08-08	56 days
9	-125	341	351	49.113717	52.604389	-436.333921	-0.071073	2019-08-08	2019-08-22	14 days
10	108	351	479	52.709597	71.081509	1984.166444	0.348550	2019-08-22	2020-02-26	188 days
11	-108	479	509	70.939346	65.179751	622.036241	0.081190	2020-02-26	2020-04-08	42 days
12	127	509	619	65.310111	117.949955	6685.260232	0.805998	2020-04-08	2020-09-15	160 days
13	-127	619	634	117.714055	115.328390	302.979473	0.020267	2020-09-15	2020-10-06	21 days
14	132	634	651	115.559047	112.009093	-468.593839	-0.030720	2020-10-06	2020-10-29	23 days
15	-132	651	663	111.785075	118.742560	-918.388066	-0.062240	2020-10-29	2020-11-16	18 days
16	117	663	675	118.980046	123.335698	509.611299	0.036608	2020-11-16	2020-12-03	17 days
17	-117	675	677	123.089026	122.127500	112.498631	0.007812	2020-12-03	2020-12-07	4 days
18	119	677	703	122.371755	130.604846	979.737891	0.067279	2020-12-07	2021-01-14	38 days
19	-119	703	710	130.343636	143.385742	-1552.010542	-0.100059	2021-01-14	2021-01-26	12 days
20	97	710	723	143.672513	134.350006	-904.283196	-0.064887	2021-01-26	2021-02-12	17 days
21	-97	723	750	134.081306	122.820000	1092.346720	0.083989	2021-02-12	2021-03-24	40 days
22	115	750	754	123.065640	120.110001	-339.898495	-0.024017	2021-03-24	2021-03-30	6 days

FIGURE 8. Trade history of the Moving Average Crossover strategy from 2018 to 2021

First of all, since this is a simulated test based on historical data, the strategy was bound to be successful, especially when experimenting on the stock price of such a lucrative company as Apple. After examining more thoroughly, one can see that most of the profit comes from a long position over the summer of 2020, which returned slightly over 80% of the invested budget. As anticipated, this explosion in the Apple stock price happened just before Apple released a line of MacOS machines operating on the Apple M1 chip.

Second of all, this strategy did not perform extremely well in a case where the price fluctuates often, which is demonstrated from row 15 to row 20. The stock price went from around 111.78 USD in

late October of 2020 to over 143 USD by the end of January 2021, which could have been a 28.82% profit in return if the bot had bought 132 shares and held them for 3 months. However, it misinterpreted the situation and kept buying and holding for shorter periods, resulting in a loss of over 10% when it attempted to go short at the price of 130.34 USD and closed the position when the stock price jumped to above 143 USD per share after less than two weeks. The bot also attempted to go short two other times in the aforementioned period, but it failed to bring about any considerable profit.

Apparently, the bot does not perform very well in the case described above because it was taking advantage of only a pair of lagging indicators, or in other words, technical indicators which only confirm the trend based on movements in the past, or sometimes, the momentum of the price increase. This is not as effective when used in a volatile market because the current price movement is much less correlated to its historical records, making it harder for the program to make its trading decision. Understandably, the next step to improve the performance of this program was to create a Machine Learning model to predict the stock price and make decisions accordingly. Theoretically, this can compensate for the weakness of lagging indicators mentioned above.

### **5.3.2 Machine Learning for classification**

Due to the arbitrary nature of stock price movement, it is nearly implausible to predict the exact value of the price at a certain moment. Therefore, this regression task was advised to be simplified to a classification one, which is to predict if the stock price goes up, down or remain the same in the near future, allowing the bot to decide if it wants to go long or go short or hold the position respectively. The workflow in this experiment was that first, the dataset was fetched, prepared and split into a train set and a test set. The model was then trained on the train set and had its predictions validated against the test set based on a certain assessing benchmark. In this experiment, the models and their evaluation metrics were provided by a Python library called *sklearn*.

#### **Data preparation**

The dataset being used for training classifiers is the same one used above for testing the Moving Average Crossover strategy. Over the course of three years, records of the first two were used for training the model, which was then tested on that of the third year. The classification target was the

movements of the closing price of the day after the following day, given today's market data. These movements were encoded using three integers, with -1 being the stock price going down, 0 means it stays the same and 1 is for when an upward trend is anticipated. For labeling training data, the closing price of each day was compared to that in the previous two days, then was labeled with one of the above depending on its movement. The stock price was considered “unchanged” if the difference is less than 4%.

While simple and easy to be collected, this dataset has a few drawbacks:

- After a quick inspection, it was evident that the dataset was skewed, or in other words, there were not as many days in which the price did not change for more than 4% (figure 9) compared to the number of days it did change:

```
data = get_OHLC_df(orig_data)
data = label_OHLC_df(data, 2, small_change_threshold=0.004)
data
```

Date	Open	High	Low	Close	Volume
2018-04-04	39.768909	41.488656	39.742377	1.0	138422000
2018-04-05	41.626139	42.024116	41.505539	1.0	107732800
2018-04-06	41.237811	41.602020	40.569688	-1.0	140021200
2018-04-09	40.974904	41.749151	40.967668	-1.0	116070800
2018-04-10	41.727450	41.968650	41.372887	1.0	113634400
...	...	...	...	...	...
2021-03-24	122.820000	122.900002	120.070000	-1.0	88530500
2021-03-25	119.540001	121.660004	119.000000	-1.0	98844700
2021-03-26	120.349998	121.480003	118.919998	1.0	93958900
2021-03-29	121.650002	122.580002	120.730003	1.0	80819200
2021-03-30	120.110001	120.400002	118.860001	-1.0	85671900

753 rows x 5 columns

```
data.Close.value_counts()
```

```
1.0    437
-1.0    257
0.0     59
Name: Close, dtype: int64
```

FIGURE 9. A quick glance at the dataset



This is bound to have a negative impact on the performance of the model if its accuracy was the key assessment metrics, known as the Accuracy Paradox [20]. Therefore, mitigation methods must be devised early on.

- The stock price moves in an extremely unpredictable pattern. In fact, there is a school of thought called the Random Walk theory [19] which hypothesizes that the future movement of a stock market is completely independent of its historical records and any attempt to outperform the market is entailed with high risks.

### **Model evaluation**

In general, the most popular evaluation metric when it comes to classification tasks is the accuracy score, which measures how many correct predictions out of the total number of predictions a model gives. However, when the values in a dataset are not equally distributed, the model can just give its predictions to the category with the most instances to achieve a high accuracy, which is downright not optimal.

To tackle this problem, this project used another metric for model assessment, whose name is F1 score, which takes into account not only the percentage of correctly categorized instances over the total number of predictions but also the number of data points in that dataset. More sophisticated evaluation metrics can certainly be added as the F1 score presented here was only for demonstration purposes.

### **Random Forest Classification**

The *backtesting.py* package did not seem to operate very well with a trading strategy which uses a Machine Learning model. In the simulation test for position trading, even though the strategy managed to act according to the prediction of the given model, it sold everything at the end of the same day while it was supposed to hold the position for a longer period of time. After some time of researching and experimenting in a vain attempt to locate and fix the bug, it was a sensible decision to write a short for loop mimicking the testing environment to see how the model performs on historical data. In figure 10, the stock price, fetched from Yahoo Finance as a pandas DataFrame, was cleaned by dropping unwanted columns and its Close column was labelled to transform this dataset into something suitable for training a classification model.

To comprehend the algorithm being used in this experiment, one should be aware of how a decision tree works. Fundamentally, it is a classification method which recursively divides the dataset into

two not necessarily equal halves, called nodes, based on a predefined criterion. This is repeated until a node cannot be split into smaller ones or until the tree reaches certain requirements (figure 10). The model then gives predictions by following the path of the branches. For example, the tree will predict that if a person whose age is 28 and does not eat pizza, that person will most likely be fit.

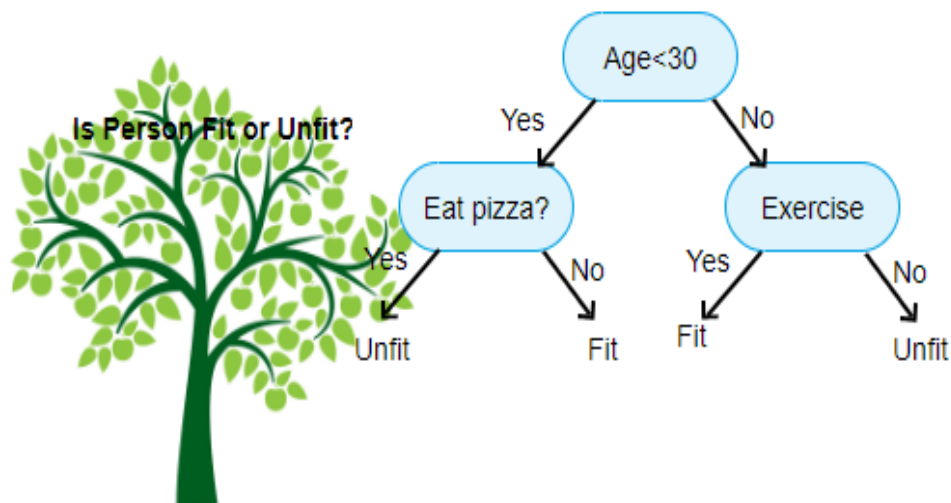


Figure 10. An illustration of how a decision tree works [21].

The sample algorithm used in this experiment was the Random Forest Classifier, which essentially is a combination of decision trees and it gives predictions by averaging all the outputs of its child individual trees. The test simulation used the market data of the year 2020 and the initial balance was set to be 4000 USD. Every time the strategy decided to buy stocks, it cannot buy more than 20% of this number. The result can be seen below (figure 11), but part of the output was clipped to save space:

```

portfolio = Portfolio()
symbol = 'AAPL'
split_date = np.datetime64('2020-03-31')
|
clf = RandomForestClassifier()
prepare_data_train_model(clf, orig_data, split_date)

for index, row in orig_data.iterrows():
    X = [row[['Open', 'High', 'Low', 'Volume']]]
    prediction = clf.predict(X)
    stock = portfolio.get_stock(symbol)
    if stock is None:
        current_price = row['Close']
        time = str(row.index[0])
        num_shares = math.floor(portfolio.max_per_stock/current_price)
        stock = None
        if prediction == 1:
            stock = Stock(symbol, current_price, num_shares, time, Position.IS_LONG, is_testing = True)
            portfolio.add_stock(stock)
        elif prediction == -1:
            stock = Stock(symbol, current_price, -num_shares, time, Position.IS_SHORT, is_testing = True)
            portfolio.add_stock(stock)
        print(f"Pred: {prediction} -- Added stock: {str(stock)} -- Total balance: {portfolio.balance}")
    else:
        if (prediction == 1 and stock.position == Position.IS_SHORT) or (prediction == -1 and stock.position == Position.IS_LONG):
            portfolio.drop_stock(stock.symbol)
            print(f"Pred: {prediction} -- Dropped stock: {str(stock)} -- Total balance: {portfolio.balance}")

portfolio.drop_stock(symbol) # sell all shares by the end to see the total return
print(portfolio.balance)

Pred: [-1.] -- Dropped stock: AAPL, 126.47101593017578, 6 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 131.77308654785156, -6 -- Total balance: 4790.638519287109
Pred: [1.] -- Dropped stock: AAPL, 131.77308654785156, -6 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 134.66876220703125, 5 -- Total balance: 3326.6561889648438
Pred: [-1.] -- Dropped stock: AAPL, 134.66876220703125, 5 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 130.6947021484375, -6 -- Total balance: 4784.168212890625
Pred: [1.] -- Dropped stock: AAPL, 130.6947021484375, -6 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 127.63926696777344, -6 -- Total balance: 4765.835601806641
Pred: [1.] -- Dropped stock: AAPL, 127.63926696777344, -6 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 136.665771484375, 5 -- Total balance: 3316.671142578125
Pred: [-1.] -- Dropped stock: AAPL, 136.665771484375, 5 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 133.7401580810547, -5 -- Total balance: 4668.700790405273
Pred: [1.] -- Dropped stock: AAPL, 133.7401580810547, -5 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 129.7100067138672, 6 -- Total balance: 3221.739959716797
Pred: [-1.] -- Dropped stock: AAPL, 129.7100067138672, 6 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 126.0, 6 -- Total balance: 3244.0
Pred: [-1.] -- Dropped stock: AAPL, 126.0, 6 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 123.98999786376953, -6 -- Total balance: 4743.939987182617
Pred: [1.] -- Dropped stock: AAPL, 123.98999786376953, -6 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 124.76000213623047, 6 -- Total balance: 3251.439987182617
Pred: [-1.] -- Dropped stock: AAPL, 124.76000213623047, 6 -- Total balance: 4000.0
Pred: [1.] -- Added stock: AAPL, 120.58999633789062, 6 -- Total balance: 3276.4600219726562
Pred: [-1.] -- Dropped stock: AAPL, 120.58999633789062, 6 -- Total balance: 4000.0
Pred: [-1.] -- Added stock: AAPL, 121.38999938964844, -6 -- Total balance: 4728.339996337891
4000.0

```

FIGURE 11. A script for testing the performance of ML model over historical data

As can be seen from the activity logs, this model does not perform very well. Some close-up observation helps with detecting a few factors which might have contributed to such low performance:

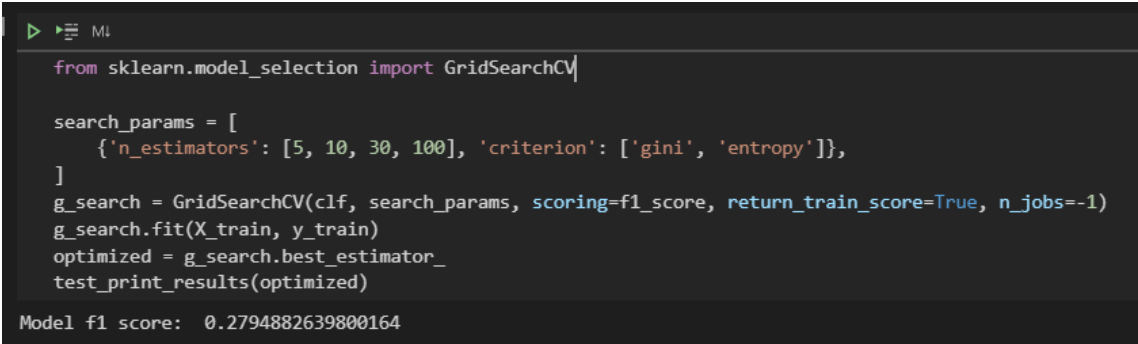
- The dataset in this scenario was a time-series, which means there was a correlation between the data points and their time index. However, the Random Forest classifier treated each datapoint equally regardless of its chronological order, which means it missed out on one of the most important features of the dataset, leading to a worse performance classifying the price movement.

- The model was quite sensitive to changes: if the price moved only as little as 4%, it considered that an indicator to action. Therefore, even if there was a slight hiccup before a big jump in price, the model took it as an invocation to sell all the stocks owned, which means the opportunity was missed. Therefore, more experiment is needed in order to find the appropriate threshold as to how much price change is considered.
- The model was also quite reactive: it acted upon the price of only the next two days, which means it did not consider a bigger picture under circumstances such as the one described above. To tackle this problem, a more sophisticated strategy and evaluation system should be devised.

### Grid search optimization

This is a method aiming to optimize the performance of Machine Learning models. What happens under the hood is that given a matrix of all necessary hyperparameters, which can be understood as the settings for a Machine Learning models such as how many loops it learns, the grid search optimizer computes all the possible combinations of the given values, then with each of them it constructs a model, trains and evaluates it. This way, the optimizer can find the best combination of configurations based on the selected evaluation metrics.

In this case, because the model being used was a multiclass Random Forest classifier, the parameter grid contained parameters such as the number of child trees and the criterion as to how a node is split. *gini* and *entropy* are two only choices provided by the *sklearn* so both of them were included here (figure 12). However, readers should consult the documentation of *sklearn* for more descriptive information as to how each of them works under the hood.



```

from sklearn.model_selection import GridSearchCV

search_params = [
    {'n_estimators': [5, 10, 30, 100], 'criterion': ['gini', 'entropy']},
]
g_search = GridSearchCV(clf, search_params, scoring=f1_score, return_train_score=True, n_jobs=-1)
g_search.fit(X_train, y_train)
optimized = g_search.best_estimator_
test_print_results(optimized)

Model f1 score: 0.2794882639800164

```

Figure 12. Random Forest classifier optimization using the Grid search method

The optimized model turned out to be even slightly worse than the one without. This concludes that there is a possibility that using only the features provided in the original dataset might not be enough, so the next step should be to enrich the training dataset with more features.

### Adding the Bollinger bands as features

The definition of this tool was provided in section 2.2.6, so only a description of the execution and result will be presented hereafter. The code implementation of the Bollinger bands is not very complicated, but to avoid wasting time reinventing the wheel, a Python package named *ta* (stands for technical indicators), was used. The moving averages of 10, 15 and 20 days were also added to the dataset, which means it had 9 predictors (column Dividends and Stock Splits were removed as they only contained the value 0) – twice as many compared to only 4 before. The code implementation and the performance of the model are presented below.

```
from ta.volatility import BollingerBands
from ta.trend import SMAIndicator

def add_lagging_indicators(df):
    bands = BollingerBands(df.Close, fillna=True)
    sma10 = SMAIndicator(df.Close, 10, fillna=True)
    sma15 = SMAIndicator(df.Close, 15, fillna=True)
    sma20 = SMAIndicator(df.Close, 20, fillna=True)
    df['BBandsHigh'] = bands.bollinger_hband()
    df['BBandsLow'] = bands.bollinger_lband()
    df['SMA_10'] = sma10.sma_indicator()
    df['SMA_15'] = sma15.sma_indicator()
    df['SMA_25'] = sma20.sma_indicator()

add_lagging_indicators(orig_data)

data = orig_data.drop(['Dividends', 'Stock Splits'], axis=1)
data = label_OHLC_df(data, period=2)

rfc = RandomForestClassifier()
search_params = [
    {'n_estimators': [5, 10, 30, 100], 'criterion': ['gini', 'entropy']},
]
g_search = GridSearchCV(rfc, search_params, scoring=f1_score, return_train_score=True, n_jobs=-1)
g_search.fit(X_train, y_train)
optimized = g_search.best_estimator_
test_print_results(optimized)

Model f1 score: 0.3064082686242362
```

Figure 13. Enriching dataset and optimizing a Random Forest

The result was slightly better, but it was still not a considerable improvement. This concludes that the Random Forest algorithm, even with an enriched dataset and being optimized, is not a

competitive candidate when it comes to time-series classification tasks such as this one. This experiment provides readers with some insights into what can be done when developing the decision-making module.

### **Scalping**

The scalping trading strategy was also included in this experiment. However, because the *backtesting.py* library did not support intraday trading, a simple testing function which loops through each entry of the data set was devised. It used the data of the previous 3 days with the recorded interval to be 1 minute and employed the Simple Moving Average Crossover strategy to make decisions. However, this did not return any resourceful insights.

### **5.3.3 Deployment phase**

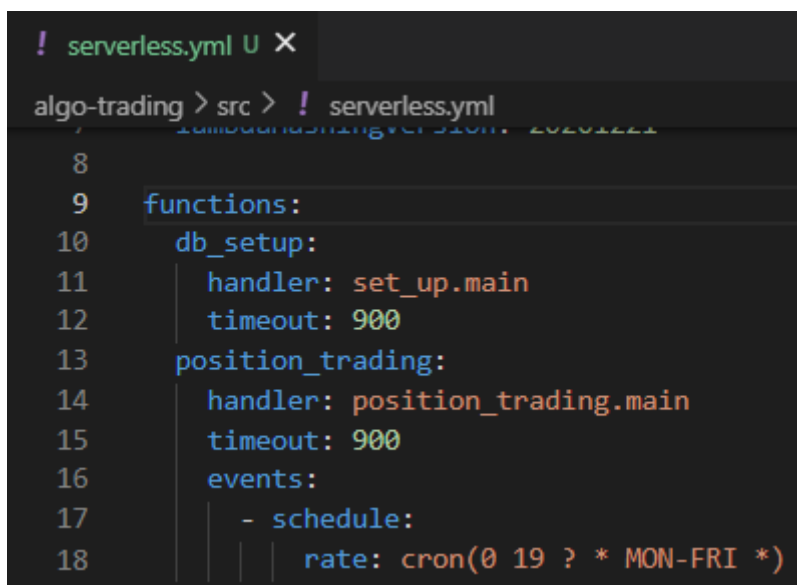
This project followed the serverless computing practice, which will be explained more thoroughly in chapter 7. The calculation for decision-making was executed in an AWS Lambda function, which pulled the source code of technical indicator classes from an AWS S3 bucket or ML models from the Mongo database depending on the settings. Because the training of Machine Learning model typically takes a considerable amount of time, this should not be carried out by Lambda, especially when the program is operating on a larger scale of data because it has a restriction of 900 seconds for each function execution and ML model training usually takes more than that in the production phase.

For demonstration purposes, the deployed Lambda function was for position trading, which made use of the Simple Moving Average Crossover strategy.

## 6 ORCHESTRATION MODULE

Another AWS service being used in this project is CloudWatch, which essentially is a resource monitoring, analytical and optimization tools for AWS services. Its CloudWatch Events feature, now is renamed Amazon EventBridge, allows users to define rules using *cron* expressions to activate certain parts of AWS services.

To do this, first one must be aware of the time zone being used in the AWS CloudWatch server so that the invocation time is correct. After some time of researching, it became clear that all scheduled events in CloudWatch Events are based on the UTC time zone[22], which led to the next question regarding what the New York market closing time is in UTC. The answer is 19.00, and since the stock exchange is only open on weekdays, the following *cron* expression was constructed and appended to the *serverless.yml* configuration file to ensure that the function is not invoked outside of the market opening hours:



```
! serverless.yml U X
algo-trading > src > ! serverless.yml
7  lambda:runtime.version: 20201221
8
9  functions:
10   db_setup:
11     handler: set_up.main
12     timeout: 900
13   position_trading:
14     handler: position_trading.main
15     timeout: 900
16     events:
17       - schedule:
18         rate: cron(0 19 ? * MON-FRI *)
```

Figure 14. Serverless configuration file of this project

In addition to that, the function timeout was set to be 900 seconds, which is the maximum allowed by Lambda to prevent premature function termination due to slow or unstable network connection because this code was run on an AWS Free Tier account.

## 7 BUSINESS LOGIC MODULE

### 7.1 Functional requirements

This module is responsible for data insertion and cascading deletion, which is to delete entries in a child table which belong to an item being deleted in the parent table. For example, when a stock is deleted in the stock table, all historical price records of it in the *daily\_price* table should be deleted as well. Upon being inserted, data should also be checked for its validity and format conformity. If these tests are not passed, it must not be inserted into the database and developers should be notified regarding the matter.

This module must also be able to fetch either historical or current market data for the decision-making module to fulfill its job. Because the process of making decisions involves a Machine Learning model whose accuracy can be improved by enriching predictors, this module also computes technical indicators from market data and passes them along to the decision-making functions if needed by the model.

To streamline the development phase as well as to make the code more readable, classes which model the trading profile of an investor in real life were also created in this module. This way, metadata such as the trading portfolio and investing balance was kept organized in one place and can be retrieved programmatically. Other data transformation and validation tasks are also implemented here.

### 7.2 Non-functional requirements

The insertion and deletion operations should not only be performed successively but also in bulk, or in other words, several transactions can be made at a time instead of having to write the insert or delete method for every entry.

The requirement for fetching data methods is that the data must be adequate, empty fields must be filled with default values and the data types and structures must be handled before persisting in the database or forwarding for further analysis.



## 7.3 Development and deployment

### 7.3.1 Database interaction methods

The first thing to do was to write methods to interact with databases. They include the following:

- A login function to get the connection to databases as well as the cursor for SQL database. In the early stages, this function was written to invoke another function which parses the credentials to log in to the database system from a configuration file, then establishes a database session.
- Methods to insert and delete entries from each table, with cascade deletion operation when needed. The database cursor object provided by the *mysqlclient* object supports a very sufficient way of bulk inserting, which involves using %s as a placeholder in the query (figure 15). To achieve cascade deletion tasks, the script first looks for all the records in the child table with a reference to the row in the parent table which is about to be deleted, then removes all of them.

```
csr = cursor if isinstance(cursor, Cursor) else connect_as_user().cursor()
query = insert_daily_price_query(securityID, data_vendorID, '%s', '%s', '%s', '%s', '%s', '%s')
number_of_rows = csr.executemany(query, data)
return number_of_rows
```

FIGURE 15. Bulk insert operation using *mysqlclient* database cursor.

Fortunately, the *mysql-connector-python* provided methods with the same name and syntax, which is highly convenient because rewriting a large part of the codebase can be avoided when changing the code dependencies.

- After having such methods ready, the next thing to do was to write a script to set up the database schema and insert default values so that the program will not crash because of a null object reference. As mentioned above in chapter 4, this only involved looping through a list of *CREATE TABLE* as well as inserting the initial balance and currency of the trading portfolio.

### 7.3.2 Data preparation methods

This involves functions for fetching, cleaning and enriching market data to train Machine Learning models. The source of data is Yahoo Finance, whose API open for the public has now been

deprecated, so a substitution being used was the *yfinance* Python package. Under the hood, this is basically a web scraper which fetches data from the website of Yahoo Finance and returns it as a DataFrame – a tabular datatype provided by the *pandas* Python library.

After being fetched, the DataFrame should contain at least the following columns: Open, High, Low, Close and Volume. These were the main columns and the irrelevant ones were dropped. Based on the configurations, each row in the *Close* column was labeled with 1, 0, and -1 if the price increased, remained unchanged or decreased compared to its 2 previous entries. By default, the movements were only recognized if the change is bigger than 4%.

During an experiment with the Random Forest classifier, the initial result led to a suggestion that the dataset should be enriched with more relevant data, which raised the need to write a function to compute the Bollinger bands values and add them to the original dataset. This used an existing library named *ta*, whose usage was described in chapter 5. Instead of providing functions for computing the indicators, this library uses a discrete class for each of them, so a little time needed to be spent on examining the documentation to use this package properly.

### 7.3.3 Program data and configuration loading

Beside the market data needed for analytical purposes, there was also a requirement to fetch data relating to the operations of the program as well as its settings. Initially, a set of methods for loading the trading profile was created. As explained in chapter 4 about the database, the current implementation of this program supports only one trading profile at a time, whose information was stored in the table named *metadata* in the SQL database. The job of such methods is to load the trading portfolio data such as the currently employed trading strategy, remaining balance and its currency from the SQL database as well as a list of stocks in possession in the Mongo database. For the program to be able to interact with these data storages, functions for reading the configuration file to fetch user credentials or connection strings were also added. They were explained in depth in section 4.4.1.

The list of stocks persisted in the Mongo database was a collection of binarized objects, so they need to be decoded using the *pickle* package before they can be used in the code. The decoding task was done as follows:

```

def fetch_stocks(db_conn = None) -> list:
    conn = object()
    stock_collection_name = 'owned_stocks'
    if isinstance(db_conn, Collection):
        if db_conn.name == stock_collection_name:
            conn = db_conn
    else:
        conn = get_remote_client(stock_collection_name)

    res = conn.find({})
    return [pickle.loads(doc['stock_bin']) for doc in res]

```

Figure 16. Function for fetching and parsing stocks from the database

After the trading profile was loaded, the next step was to load the decision-maker, based on the value of row *current\_strategy* in the *metadata* table. As mentioned above, there were two types of strategies being experimented with, so if the strategy name starts with “ML\_”, it means that the decision-maker is a Machine Learning model, which should be found in the collection named *models* in the Mongo database. When queried with the Python code, this collection is automatically created if it does not exist.

#### 7.3.4 Model classes

To ease up the process of loading the trading profile, two object-relational management classes were created with the name Portfolio and Stock. They contain attributes with the same names as they are in the database tables as well as other methods for constructing their instance and saving changes if necessary.

#### 7.3.5 Deployment phase

##### Serverless computing

This project aims to achieve a serverless computing model, which essentially means that the application does not depend on a constantly working computer server but a set of computing functions, hosted by a cloud provider and invoked by predefined events. In this way, the developer does not have to take care of backend development, routing, provisioning or server maintenance and can focus more on improving his application. It is also economical in the sense that the

resources are only allocated when needed to save money on running physical servers or keeping a virtual machine alive on a cloud platform.

For this purpose, AWS Lambda was chosen to be the computing service to host the decision-making function because of its free access and large user community. Thanks to the large AWS ecosystem, it can be easily incorporated with other services required for the program to run smoothly such as database systems and orchestration services. The workflow was as follows:

1. An AWS CloudWatch rule is created to automatically invoke the Lambda function at a certain time of the day. For position trading, this will be a short while before the market closes.
2. Upon being activated, the Lambda function first pulls metadata such as portfolio balance and list of stocks in possession from AWS Relational Database System (RDS) and Mongo Atlas
3. The function then fetches market data of that day via the API provided by Yahoo Finance
4. Decision regarding whether to buy, sell or keep holding the position will be computed based on the strategy or the prediction of the Machine Learning model fetched in step 2.
5. For the sake of experimentality, this program will only make a pseudo transaction, which is to add the acquired shares to the portfolio and calculate the remaining balance accordingly without placing an actual order to an online broker.
6. The state of the trading profile as well as the information of the mock transaction are then saved to the appropriate database.

### **Packaging source code and dependencies**

Because the official guide provided by AWS on how to deploy code on Lambda requires a bit more effort of tweaking, a framework named Serverless was used for this purpose. With Serverless, several Lambda functions of the same project which have some dependencies in common can be declared within the same Serverless configuration file, making it faster to deploy functions to the server of AWS Lambda.

There is a common error when using Serverless to deploy Python code to Lambda, which is an error with character encoding in the *requirements.txt* file required for every Python project to install the correct version of its dependencies. For unknown reasons, the framework can only read ANSI-encoded .txt file, while the *pip freeze* command typically outputs the value in UTF encoding. For a Linux-based development environment, this can be solved easily using a program named *iconv*,

which can be appended to the *pip freeze* command to encode text before saving it to the file. However, since this project was developed in a Windows environment, the solution was to use the *Notepad* program to change the encoding of the file *requirements.txt* manually before deployment. Another solution could be to write a Python function which converts the *stdout* into the desired encoding.

When deploying the function to Lambda, it is also strictly limited that the total size of the unzipped code is below 250 MB. Therefore, when using the Serverless framework to perform this task, the value of *zip* and *slim* should be set to *true*. This essentially means that all the binary or temporary files such as those with *.pyc* format are removed and the whole source code of the project was compressed before uploading to AWS S3.

This phase was also the most time-consuming of this project because many of the dependencies, especially the *mysqlclient* and *numpy* packages, rely on their compiled C binaries to function normally. This introduced two problems if not the project is not packaged correctly:

- The main Lambda function cannot be invoked without errors because the dependencies are not compiled.
- If the compiled dependencies are included in the upload, it would exceed the size limit allowed by Lambda, resulting in a failed operation.

The solution employed in this project will be described hereafter.

## Deployment process

With the Node package *serverless* installed globally, the deployment began with running the command *serverless create* with a value *aws-python3* passed as an argument. This created a *serverless.yml* configuration file as well as other necessary files and a sample file *handler.py* as the default handler to be set as a Lambda function.

The code of this module was uploaded to an S3 bucket, which was executed by the Lambda function when invoked. There were two Lambda functions to be deployed: *db\_setup* and *position\_trading*, which resets the database and computes the daily trading decisions respectively. The Python connector initially selected to interact with SQL database, *mysqlclient*, turned out to require the command *mysql\_config* to be installed on the host system as well as its C package to be compiled prior to function invocation, which extends considerably the size of the deployment package. Therefore, a search for a better contender which is written purely in Python was needed.

The tutorial on AWS recommends using a package named *pymysql* [23], but even that turned out rely on the *cryptography* package, which can be a hassle to install properly. Therefore, the final choice, though should have been more obvious from the beginning, was to use the package *mysql-connector-python* provided by MySQL itself.

There was another step to execute before invoking the Lambda function, which was to attach it to the same VPC which hosts the RDS instance. This is because, as mentioned above, the instance was set to be private to prevent it from malicious users. By being attached to the same VPC, the Lambda function was automatically granted access to the RDS instance.

However, this introduced another problem: the Lambda function needs to pull data from a Mongo cluster in a different VPC, but because it had been attached to a VPC, it was isolated from such connections. This situation resorted to a method called VPC peering connection, which is a network connection which allows private traffic routing between two VPCs [24] and acts as a secured bridge for the data request resulted from Lambda function invocation to travel to the VPC in which the Mongo node resides. To set up this kind of connection requires solid knowledge in networking and security, so it took a short while to have everything set up properly.

### 7.3.6 Results

The source code and its dependencies were successfully uploaded to an S3 bucket, which was generated automatically by the Serverless framework. A new AWS CloudFormation stack was also created to manage the resources needed for the Lambda function to execute successfully.

Before the functions were actually uploaded to AWS, they were tested locally using the command:

```
serverless invoke local -f <function-name>
```

It is crucial that this command is run within the same directory with the serverless configuration file, otherwise, the program will throw an error saying that it cannot find the config file, which also means that it cannot find the function with the given name. When invoked locally, both functions performed fine, which was likely because they used the already-compiled resources available in the

development environment. However, this was not the case when invoking them as Lambda functions.

When the Lambda function *db\_setup* was invoked on the cloud, it performed the task without errors, although it seemed to take somewhat more time than when being executed locally. The other function, *position\_trading*, was also executed successfully, but this outcome was not very consistent. The script would usually throw an exception with the name `ServerSelectionTimeout` error. The initial hypothesis was that the IP of the Lambda executor was not added to the inbound traffic list of the Mongo cluster, but this was proven to be wrong because even after exposing the Mongo cluster to the public by setting the inbound rule to be `0.0.0.0/0`, it would still throw this error. It turned out that the IP of a Lambda executor is dynamic, meaning that it changes for every runtime because the executor is routed to the most available zone at runtime [25]. After a considerable amount of time on researching, the second diagnosis was that the Lambda function was isolated from the Internet after having been attached to the VPC so it cannot reach the Mongo cluster, which resides in another VPC. Establishing a VPC peering connection fixed this error to some extent because the Lambda execution was successful but not very consistently. Therefore, the possible cause of this error was narrowed down to the AWS resource availability, most likely because of the Free Tier account [26].

A useful feature of AWS Lambda is that it is inherently integrated with AWS CloudWatch Logs, which typically contains outputs of the function as well as the stack trace. When ran successfully, a message is displayed in the test interface of Lambda, saying that the result of this execution is successful. It can also be seen from the screenshot below that this function is linked to a CloudWatch Event, which was described in chapter 6:

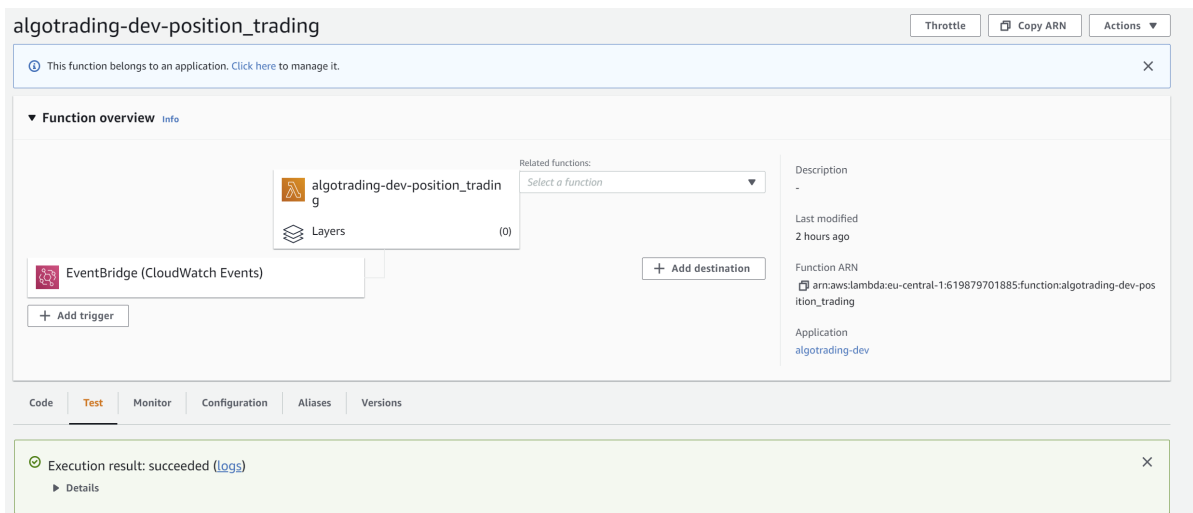


Figure 17. Successful execution message of Lambda

A more descriptive log can be found in the log groups of CloudWatch (figure 17), which is easily accessible by clicking the “logs” hyperlink in the message box above. In this case, because Apple’s stock price did not have much going on at the time of execution, the function simply just saved the portfolio and exited.

CloudWatch > CloudWatch Logs > Log groups > /aws/lambda/algotrading-dev-position\_trading > 2021/05/07/[\$LATEST]540c3b7324fb44bab9e3778e68f36595

**Log events**  
You can use the filter bar below to search for and match terms, phrases, or values in your log events. [Learn more about filter patterns](#)

Q Filter events

Timestamp	Message
	No older events at this moment. <a href="#">Retry</a>
2021-05-07T10:12:20.427+03:00	START RequestId: 022117a3-d719-4ccd-98fa-9842f156ff18 Version: \$LATEST
2021-05-07T10:12:37.289+03:00	OpenBLAS WARNING - could not determine the L2 cache size on this system, assuming 256k
2021-05-07T10:12:40.282+03:00	/tmp/sls-py-req/joblib/_multiprocessing_helpers.py:45: UserWarning: [Errno 38] Function not implemented. joblib will operate in serial mode
2021-05-07T10:12:40.282+03:00	warnings.warn('%s. joblib will operate in serial mode' % (e,))
2021-05-07T10:12:40.483+03:00	OpenBLAS WARNING - could not determine the L2 cache size on this system, assuming 256k
2021-05-07T10:12:42.170+03:00	Accessing config file from /var/task
2021-05-07T10:12:42.170+03:00	algotrader1 somethingmysterious
2021-05-07T10:12:42.170+03:00	Accessing config file from /var/task
2021-05-07T10:12:42.209+03:00	Accessing config file from /var/task
2021-05-07T10:12:42.252+03:00	Loading portfolio ...
2021-05-07T10:12:42.255+03:00	Accessing config file from /var/task
2021-05-07T10:12:42.373+03:00	Portfolio loaded.
2021-05-07T10:12:42.373+03:00	Loading TA class...
2021-05-07T10:12:42.373+03:00	TA class loaded.
2021-05-07T10:12:42.373+03:00	Loading market data from Yahoo Finance...
2021-05-07T10:12:42.739+03:00	[ 0% ] [*****67%*****] 2 of 3 completed [*****100%*****] 3 of 3 completed
2021-05-07T10:12:42.740+03:00	Market data loaded.
2021-05-07T10:12:42.740+03:00	Evaluating price and making decisions:
2021-05-07T10:12:42.751+03:00	Saving portfolio...
2021-05-07T10:12:42.763+03:00	Accessing config file from /var/task
2021-05-07T10:12:42.891+03:00	Portfolio saved to database.
2021-05-07T10:12:42.892+03:00	END RequestId: 022117a3-d719-4ccd-98fa-9842f156ff18
2021-05-07T10:12:42.892+03:00	REPORT RequestId: 022117a3-d719-4ccd-98fa-9842f156ff18 Duration: 22464.62 ms Billed Duration: 22465 ms Memory Size: 1024 MB Max Memory Used: 750 MB
	No newer events at this moment. <a href="#">Auto retry paused.</a> <a href="#">Resume</a>

Figure 18. A detailed log of a successful Lambda invocation



## 8 CONCLUSION

Overall, the initial goal was achieved, which is to have a working prototype of an algorithmic trading bot deployed to the AWS cloud services, with all components successfully deployed to their designated locations. The CloudWatch scheduler worked as anticipated and its activity logs are recorded properly. The database clusters were working consistently and securely. The strategies using technical indicators were proved to be highly profitable over a long period of time, although when employing a sample Machine Learning model, it was not as closely successful. However, this is tolerable since optimizing such models, especially when it comes to such volatility experienced in the stock market, is definitely not a trivial and overnight task.

Admittedly, the Lambda invocation did not have a high success rate due to limited network performance granted to the AWS Free Tier account. The progress was also stagnated in the late stage due to deployment-related errors caused by dependencies incompatibility between the development and deployment environments. Therefore, it is a good practice to have decided the cloud platform to use well before the code implementation of the program so that the chosen dependencies work well with the deployment environment.

It is also worth noticing that creating a profitable Machine Learning model is an enormously demanding task. As demonstrated above, creating a strategy which only acts upon the two-day prediction of a model simply cannot produce a good result because the stock market is extremely volatile and requires the strategy to be able to take into account the big picture. Readers should therefore consider this as an educational example instead of a production-level program to be deployed and exploited.

The current implementation also allows significant further development in the future. The most evident one is to improve the predictive performance of the strategy. This can be done using the Facebook Prophet model for example, which performs smoothly on Linux-based systems. Additionally, there are only two workflows defined as Lambda functions currently, which are responsible for setting up the new database cluster and performing the decision-making process for position trading, so another Lambda function which invokes the process of re-training Machine Learning models can be developed.

Given more computing resources, one can scale up the frequency of function invocation to make the program capable of intraday trading, which requires the function to be executed thousands of times instead of just one. However, the trading strategies for such usage will also need to be devised thoroughly because position trading strategies do not have the capability to take advantage of minor price changes as shown in the experiment in chapter 5.

This project is also a good starting point to learn about the serverless computing as well as how it can contribute to one of the most important tasks which human has to face in a daily situation: decision making by examining a typical program of this type – an algorithmic trading bot. Readers can also have a general understanding of the most popular AWS services such as AWS Lambda, AWS S3, AWS CloudWatch and AWS RDS as well as how they all work together to deliver a serverless cloud application. Such knowledge is bound to be useful in the working life for graduate students.

## REFERENCES

1. J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh and A. H. Byers, "Big data: The next frontier for innovation, competition and productivity", 2011. [Online PDF] Available:  
[https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI\\_big\\_data\\_exec\\_summary.pdf](https://www.mckinsey.com/~media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_exec_summary.pdf) [Accessed Mar. 26, 2021]
2. The Forbes, "The Forbes 400", 2020. [Online]. Available: <https://www.forbes.com/forbes-400/> [Accessed Mar. 26, 2021]
3. W. Kenton, "Financial Instrument" in Investing Essentials. Investopedia, [online document] 2020. Available: Investopedia, <https://www.investopedia.com/terms/f/financialinstrument.asp> [Accessed Mar. 2, 2021]
4. W. Kenton, "Security" in Investing Essentials. Investopedia, [online document] 2020. Available: <https://www.investopedia.com/terms/s/security.asp> [Accessed Mar. 2, 2021]
5. A. Hayes, "Trade", in Economics. Investopedia, [online document] 2021. Available: <https://www.investopedia.com/terms/t/trade.asp> [Accessed Mar. 2, 2021]
6. W. Kenton, "Exchange" in Investing Essentials. Investopedia, [online document] 2020. Available: <https://www.investopedia.com/terms/e/exchange.asp> [Accessed Mar. 2, 2021]
7. T. Smith, "Broker" in Brokers. Investopedia, [online document] 2020. Available: <https://www.investopedia.com/terms/b/broker.asp> [Accessed Mar. 2, 2021]
8. J. Chen, "Trading Strategy" in Trading basic education. Investopedia [Online] 2019. Available: <https://www.investopedia.com/terms/t/trading-strategy.asp> [Accessed: Mar. 3, 2021].
9. J. Chen, "Technical Indicator", 2021. [Online]. Available: <https://www.investopedia.com/terms/t/technicalindicator.asp> [Accessed: Mar. 29, 2021]
10. Babypips, "The beginner's guid to FX trading", *babypips.com*. [Online]. Available: <https://www.babypips.com/learn/forex/using-moving-averages> [Accessed: Mar. 31, 2021].
11. J. Chen, "Take-Profit Order – T/P", 2020. [Online]. Available: <https://www.investopedia.com/terms/t/take-profitorder.asp> [Accessed: Apr. 6, 2021]
12. M. Kramer, "Stop-Lost Order", 2020. [Online]. Available: <https://www.investopedia.com/terms/s/stop-lossorder.asp> [Accessed: Apr. 6, 2021]

13. A. Hayes, "Bollinger Band®," Investopedia, 30-Apr-2021. [Online]. Available: <https://www.investopedia.com/terms/b/bollingerbands.asp>. [Accessed: 11-May-2021].
14. C. Mitchell, "Oversold Definition," Investopedia, 11-Dec-2020. [Online]. Available: <https://www.investopedia.com/terms/o/oversold.asp>. [Accessed: 11-May-2021].
15. "Virtual Environments and Packages" Python Software Foundation - Python 3.9.5 documentation. [Online]. Available: <https://docs.python.org/3/tutorial/venv.html>. [Accessed: 11-May-2021].
16. "Docker overview," Docker Documentation, 10-May-2021. [Online]. Available: <https://docs.docker.com/get-started/overview/#the-underlying-technology>. [Accessed: 11-May-2021].
17. P. Sanghvi, "Selecting a Database for an Algorithmic Trading System", *Medium*, 2019. [Online]. Available: <https://medium.com/prooftrading/selecting-a-database-for-an-algorithmic-trading-system-2d25f9648d02> [Accessed Mar. 30, 2021]
18. A. Rubin, "Column Store Database Benchmarks: MariaDB ColumnStore vs. ClickHouse vs. Apache Spark", *Percona*, 2017. [Online]. Available: <https://www.percona.com/blog/2017/03/17/column-store-database-benchmarks-mariadb-columnstore-vs-clickhouse-vs-apache-spark/> [Accessed: Mar. 30, 2021]
19. "Random walk theory definition," IG. [Online]. Available: <https://www.ig.com/en/glossary-trading-terms/random-walk-theory-definition>. [Accessed: 11-May-2021].
20. T. Afonja, "Accuracy Paradox," *Medium*, 10-Dec-2017. [Online]. Available: <https://towardsdatascience.com/accuracy-paradox-897a69e2dd9b>. [Accessed: 11-May-2021].
21. A. Chavan, "A Comprehensive Guide to Decision Tree Learning," *AI TIME JOURNAL*, 17-Jan-2020. [Online]. Available: <https://www.aitimejournal.com/@akshay.chavan/a-comprehensive-guide-to-decision-tree-learning>. [Accessed: 11-May-2021].
22. "Schedule Expressions for Rules - Amazon CloudWatch Events". Amazon Web Services Inc. [online] Available at: <https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/ScheduledEvents.html> [Accessed 11 May 2021].
23. R. W. Hendrix, "Lambda" Amazon Web Services Inc.. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/services-rds-tutorial.html>. [Accessed: 11-May-2021].

24. "VPC peering", Amazon Web Services Inc., 2016. [Online]. Available: <https://docs.aws.amazon.com/vpc/latest/userguide/vpc-peering.html>. [Accessed: 11-May-2021].
25. M. Leak, "AWS Lambda: Under the Hood-How Lambda Works," *Medium*, 20-Jan-2021. [Online]. Available: <https://matthewleak.medium.com/aws-lambda-under-the-hood-how-lambda-works-43efba14d899>. [Accessed: 11-May-2021].
26. L. Saarelainen, personal message in Slack. (May 11<sup>th</sup>, 2021).