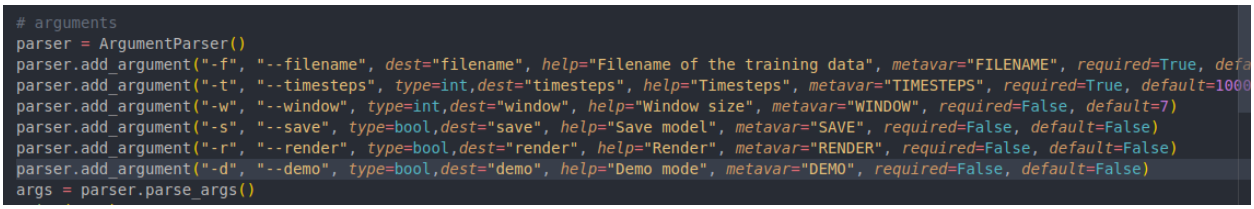
Production Requirements

The following are the production requirements for the upcoming project:

Input Argument Requirements

What are the arguments that your code needs during execution? This critic because we don’t want to change the code by hand at every execution.

An example is provided below.



```
# arguments
parser = ArgumentParser()
parser.add_argument("-f", "--filename", dest="filename", help="Filename of the training data", metavar="FILENAME", required=True, defa
parser.add_argument("-t", "--timesteps", type=int,dest="timesteps", help="Timesteps", metavar="TIMESTEPS", required=True, default=1000
parser.add_argument("-w", "--window", type=int,dest="window", help="Window size", metavar="WINDOW", required=False, default=7)
parser.add_argument("-s", "--save", type=bool,dest="save", help="Save model", metavar="SAVE", required=False, default=False)
parser.add_argument("-r", "--render", type=bool,dest="render", help="Render", metavar="RENDER", required=False, default=False)
parser.add_argument("-d", "--demo", type=bool,dest="demo", help="Demo mode", metavar="DEMO", required=False, default=False)
args = parser.parse_args()
```

Preprocessing

What is the preprocessing stage that should be executed once before the main simulation loop.

The preprocessing stage should include tasks such as data cleaning, data normalization, and data augmentation. These tasks are important to ensure that the data is ready for analysis and modeling. Additionally, any necessary data transformations or feature engineering should also be done during this stage.

An example is provided below.

```python
if __name__ == "__main__":
    tprint("Starting live trading...")
    # get initial prices
    tprint("Getting initial prices... ~ 30sec")
    initPrices = getInitialPrices(120)
    df = pd.DataFrame(initPrices, columns=['Close'])
    tprint("Initial prices received.")
    tprint("Loading model...")
    # load the model and run it
    model = A2C.load(find_files(VERSION))
    live_env = CryptoEnv(df=df, window_size=10, frame_bound=(10, 1440*4))
    obs = live_env.reset()
    while True:
        obs = obs[np.newaxis, ...]
        action, _states = model.predict(obs)
        obs, rewards, done, info = live_env.step(action)
        if done:
            tprint("Model loaded. Past 30 minute trading simulation done.")
            break

    # Create a trader object
    tprint("Creating trader object...")
    CA = CryptoArsenal(df)

    # get ready for the continuous trading
    history = []
    minute = 1
```

Main Loop

What code should work in the main loop while trading? An example is provided below.

```
while True:
    try:
        # Get the latest price and current time
        price, price_status = getCurrentPrice()
        if price_status == 200:
            pass
        else:
            tprint("Error getting price. Retrying...")
            continue
        # Update the environment with the latest price
        live_env.df = live_env.df.append({'Time':int(time()),'Close': price}, ignore_index=True)
        live_env.prices, live_env.signal_features = live_env._process_data()
        obs = live_env._get_observation()

        # Update the trader object
        #CA.update(price)

        # Take a step in the environment with the current model
        obs = obs[np.newaxis, ...]
        action, _states = model.predict(obs)
        obs, rewards, done, info = live_env.step(action)

        response, sentAction, ID = CA.send_webhook_request(action[0],price)
        orders = ["Sell", "Buy"]
        tprint(f"[{ID}] :: Price: {price} | Action: {sentAction} | Signal: {orders[int(action[0])]} | \
               Webhook log: {response} | Model Parameter: {round(live_env._total_profit,4)}")
        # History handle
        info["price"] = price
        history.append(info)
        if demo:
            live_env.render(dynamic=True)
        # Wait 1 minute
        if minute%10 == 0:
            # get summary
            CA.get_summary()
            # Save the history
            pd.DataFrame(history).to_csv(f"history/live/hist.csv")

        minute += 1
        sleep(60)
```

There is a trading class in legacy I am providing the source code. This is only the minimum working code.

```
import requests
from creds.keys import crypto_arsenal_payload


class CryptoArsenal:
    def __init__(self):
        self.URL = 'https://api.crypto-arsenal.io/trading-signal/webhook'

    def send_webhook_request(self, position: str, orderId: str):
        """
        :param position: open_long, close_long, open_short, close_short, cancel_all
        :param orderId: exchange id
        :return: response
        """
        payload = crypto_arsenal_payload[position]
        payload["clientOrderId"] = orderId
        response = requests.post(self.URL, json=payload)
        return response.text
```

This can be extended to a trader object such as:

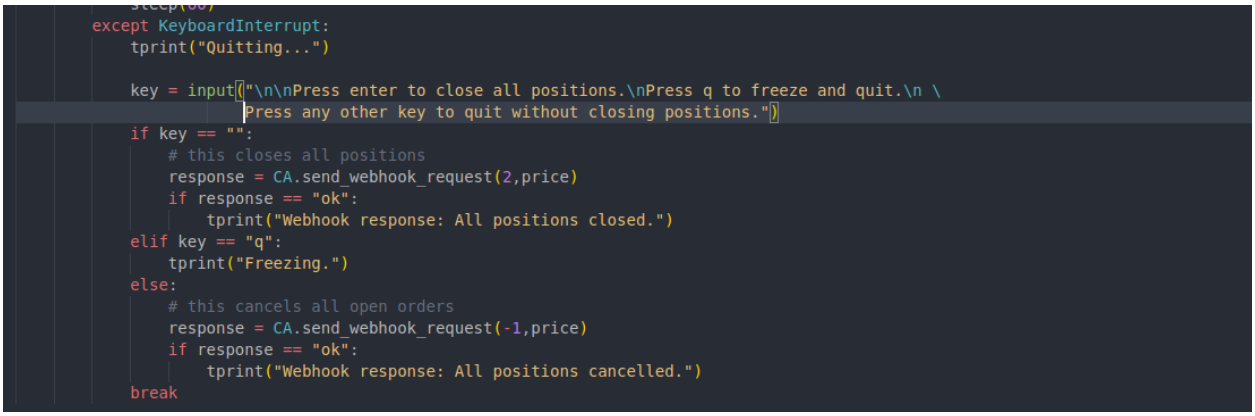
```
class CryptoArsenal:
    def __init__(self,df):
        self.URL = 'https://api.crypto-arsenal.io/trading-signal/webhook'
        self.TOKEN = "25df6234-1227-4a45-afc2-30712e87506c" #"25df6234-1227-4a45-afc2-3071
2e87506c"
        self.NAME  = 'RLBot-TV-13152' #"RLBot-TV-13152"
        self.TRADE_SIZE = "0.001"
        self.pf = 0.0
        self.pfwf = 0.0
        self.df = df
        self.df['20_EMA'] = self.df['Close'].ewm(span = 20, adjust = False).mean()
        self.df['50_EMA'] = self.df['Close'].ewm(span = 50, adjust = False).mean()
        self.df['Signal'] = np.where(self.df['20_EMA'] > self.df['50_EMA'], 1.0, 0.0)
        self.df['Position'] = self.df['Signal'].diff()
        print(self.df.tail(5))
        # @TODO add utility to read previous longs and shorts from file
        self.opened_longs = []
        self.opened_shorts = []
        self.number_of_trades = 0
        longFile = pd.read_csv("history/live/longs.csv")
        shortFile = pd.read_csv("history/live/shorts.csv")
        for i in range(len(longFile)):
            self.opened_longs.append((int(longFile['id'][i]),float(longFile['price'][i])))
            self.number_of_trades += 1
        for i in range(len(shortFile)):
            self.opened_shorts.append((int(shortFile['id'][i]),float(shortFile['price']
[i])))
            self.number_of_trades += 1
        del longFile,shortFile

    def send_webhook_request(self, signal:int, price: float):
        """
        :param position: open_long, close_long, open_short, close_short, cancel_all
        :param orderId: exchange id
        :return: response
        """
        orderId = str(int(time())).replace('.0','')

        status = "No trading"
        action = "No action"
        trading = False
        if signal == 0:
            profits = self.get_max_profit_among_longs(price)
            if len(profits) != 0:
                for profit,item in profits:
                    payload = payloads['close_long']
                    payload["clientOrderId"] = item[0]
                    response = requests.post(self.URL, json=payload)
```

On Quit

On quit, how does your program behave? An example is provided below.



```
        except KeyboardInterrupt:
            tprint("Quitting...")

            key = input("\n\nPress enter to close all positions.\nPress q to freeze and quit.\n \
                        Press any other key to quit without closing positions.")
            if key == "":
                # this closes all positions
                response = CA.send_webhook_request(2,price)
                if response == "ok":
                    tprint("Webhook response: All positions closed.")
            elif key == "q":
                tprint("Freezing.")
            else:
                # this cancels all open orders
                response = CA.send_webhook_request(-1,price)
                if response == "ok":
                    tprint("Webhook response: All positions cancelled.")
            break
```

Data Gathering

Working with xlsx file is costly we should switch to csv. According to internet:

> CSV files are also easier to work with programmatically since they can be read as text files. Xls files, on the other hand, are binary files that need to be parsed by a special library in order to be read.

Difference	Xls/Xlsx Files	CSV Files
File type	Binary file holding all data in a group of worksheets	Plain text format with comma-separated data
Editing	Needs excel editors to open, cannot open in any text editors	Simple files can be opened in text editors like notepad
Formatting	It contains formulas, macros, formatting, and stores data as well	A simple file with stored data
Size	Larger in size	Smaller in size
Speed	Faster to load	Comparatively needs time to load

This is because it's always easier to add data and read data together with a sequential data rather than a binary file. We will **read and write** in the same time. Therefore it's better to use **csv** to reduce computational complexity of writing and reading.