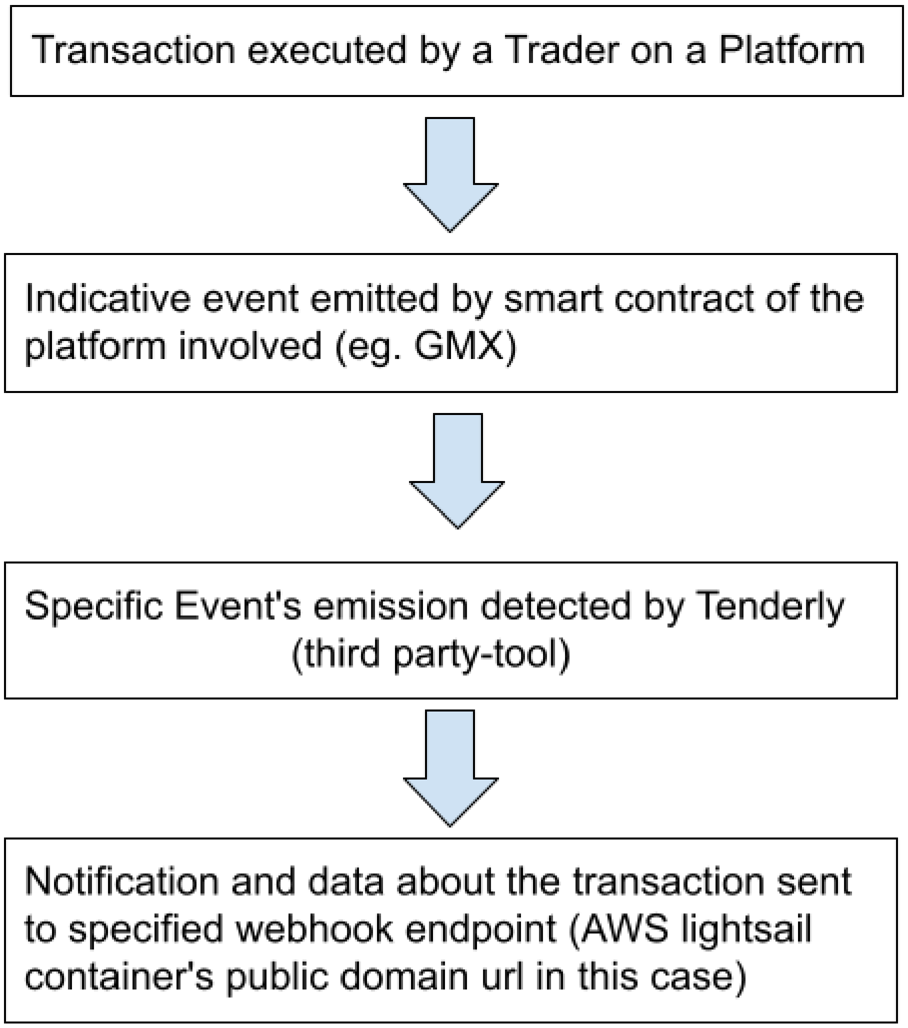
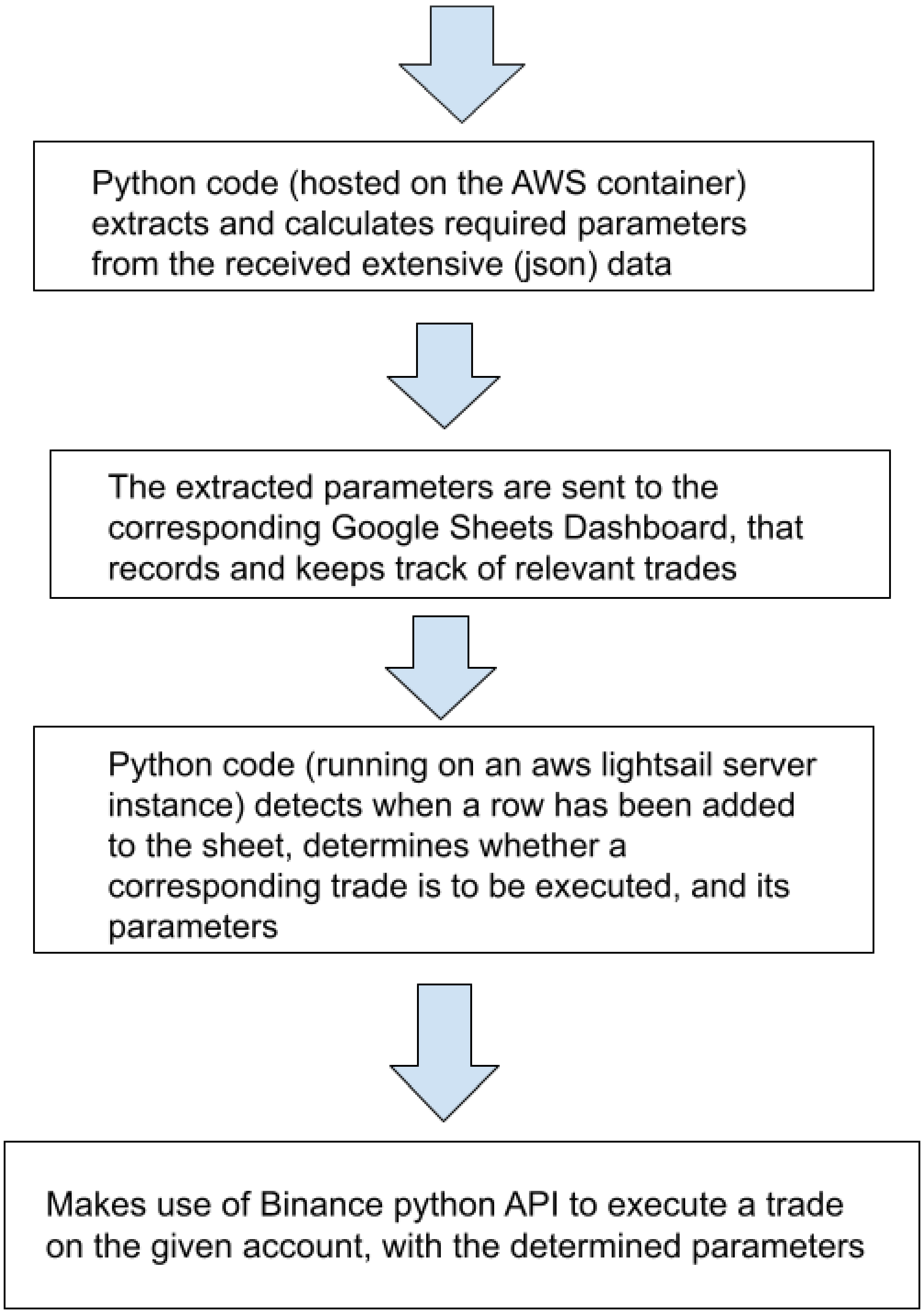
Overview of the Auto-Trading System

One of the broader objectives of the project is to mirror the trading patterns of selected profitable traders, by executing corresponding trades of one's own and thus benefitting from strategies employed by the successful traders.

Due to the frequency and varied timings of the trades, along with the need to minimize the time delay between the trade being copied and its equivalent on our account, this would have to be done automatically.

The following illustration outlines the overall flow of the current system:


Transaction executed by a Trader on a Platform
Indicative event emitted by smart contract of the platform involved (eg. GMX)
Specific Event's emission detected by Tenderly (third party-tool)
Notification and data about the transaction sent to specified webhook endpoint (AWS lightsail container's public domain url in this case)


Python code (hosted on the AWS container) extracts and calculates required parameters from the received extensive (json) data
The extracted parameters are sent to the corresponding Google Sheets Dashboard, that records and keeps track of relevant trades
Python code (running on an aws lightsail server instance) detects when a row has been added to the sheet, determines whether a corresponding trade is to be executed, and its parameters
Makes use of Binance python API to execute a trade on the given account, with the determined parameters

Note that in order to speed up trade execution by a couple of seconds (and reduce time delay in sending data to the dashboard and back), data may be sent

to the dashboard after, and not before trade execution. The trade may be executed directly after the params are extracted.
However, the current setup makes it more transparent and makes it easier to inspect early-stage data, while also decoupling the trade execution module from the extraction module.

There will be separate documentation for sub-modules of the system (on AWS, the dashboard code, Tenderly, and binance APIs)

Platforms' Smart Contracts:

DeFi trading platforms such as GMX contain smart contracts execute automatically when certain conditions are met, ie. when a certain trade or transaction is made by a wallet.

The system currently handles trades occurring on the platform GMX. Smart contracts of platforms deployed on a public blockchain are generally public.

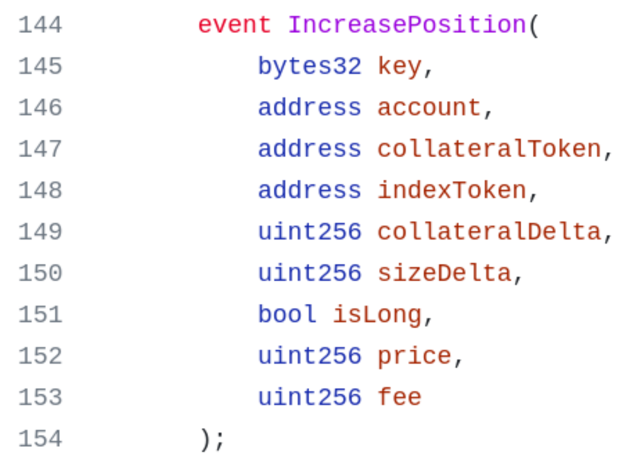
GMX's smart contracts may be found in the below link:
https://github.com/gmx-io/gmx-contracts/tree/master/contracts

Emission of events:

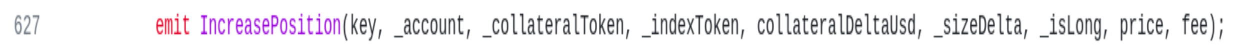
Emitting events is a method for smart contracts to communucate with external services and notify one of the current state of a contract, signalling actions that occurred.

We will be detecting certain events emitted by GMX's smart contract in order to obtain information regarding transactions occurring on the platform.

For example, we will be detecting the emission of the 'IncreasePosition' event to identify when a trader is opening/increasing a position, and the emission of a 'DecreasePosition' event to figure when a trader is partially or fully closing/decreasing their position.
Identifying the functions and significance of events requires an analysis of the smart contract's Solidity code.
For example, the 'IncreasePosition' event's definition and emission can be found in core/Vault.sol (in the above GMX contract's github link).



```
    event IncreasePosition(
        bytes32 key,
        address account,
        address collateralToken,
        address indexToken,
        uint256 collateralDelta,
        uint256 sizeDelta,
        bool isLong,
        uint256 price,
        uint256 fee
    );
```



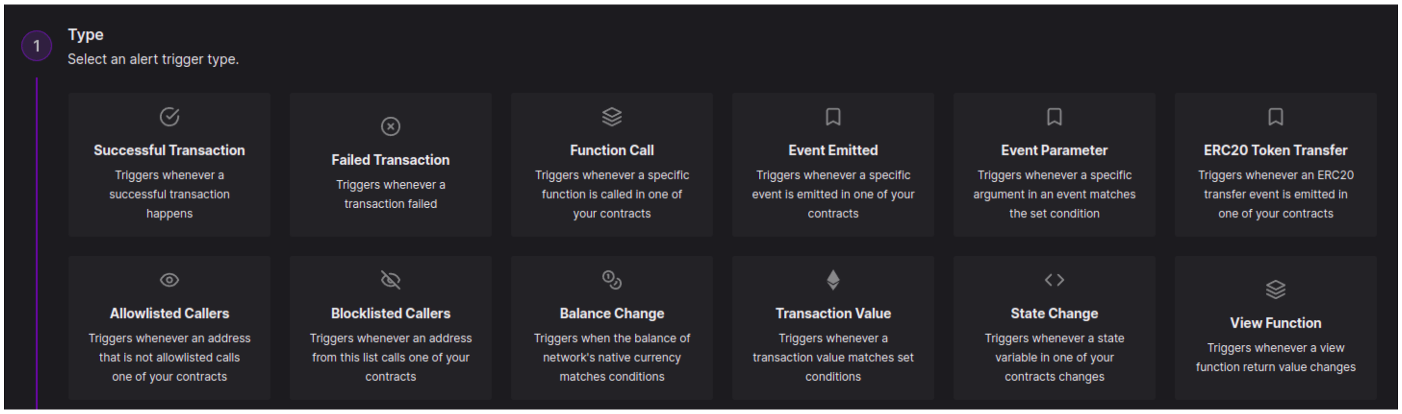
```
        emit IncreasePosition(key, _account, _collateralToken, _indexToken, collateralDeltaUsd, _sizeDelta, _isLong, price, fee);
```

Detecting emitted Events:

Now we need a mechanism to detect when a certain smart contract is emitted, in real time.

To do this, we use a third-party tool 'Tenderly', a platform that allows web3 devs to monitor smart contracts, among other things. Specifically, we use Tenderly Alerts, via which we can receive notifications regarding on-chain events in real time.
(https://tenderly.co/)

Alerts can be set for a variety of triggers



1 **Type**
Select an alert trigger type.

Successful Transaction
Triggers whenever a successful transaction happens

Failed Transaction
Triggers whenever a transaction failed

Function Call
Triggers whenever a specific function is called in one of your contracts

Event Emitted
Triggers whenever a specific event is emitted in one of your contracts

Event Parameter
Triggers whenever a specific argument in an event matches the set condition

ERC20 Token Transfer
Triggers whenever an ERC20 transfer event is emitted in one of your contracts

Allowlisted Callers
Triggers whenever an address that is not allowlisted calls one of your contracts

Blocklisted Callers
Triggers whenever an address from this list calls one of your contracts

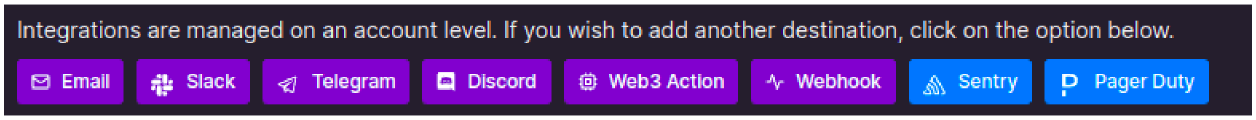
Balance Change
Triggers when the balance of network's native currency matches conditions

Transaction Value
Triggers whenever a transaction value matches set conditions

State Change
Triggers whenever a state variable in one of your contracts changes

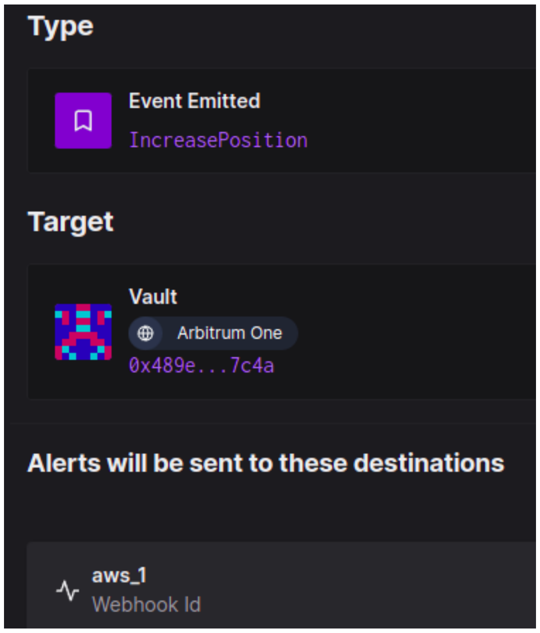
View Function
Triggers whenever a view function return value changes

Multiple destinations for the alerts may be set:



Integrations are managed on an account level. If you wish to add another destination, click on the option below.

There will be separate specific documentation regarding Tenderly and setting up alerts.
Fow now, the following is an alert set up to receive notifications and data when the discussed 'IncreasePosition' event is emitted, and sends the data to a webhook endpoint.



Type

Event Emitted
IncreasePosition

Target

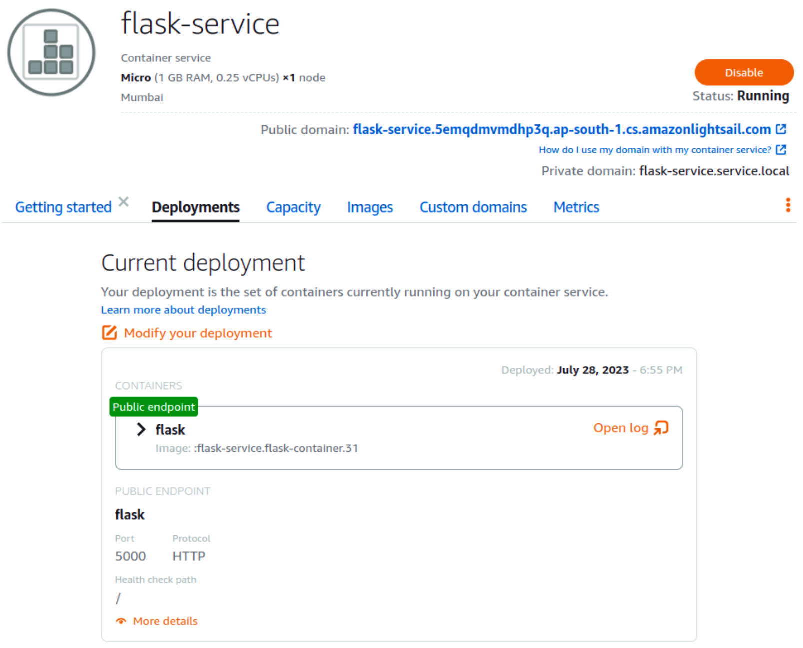
Vault
Arbitrum One
0x489e...7c4a

Alerts will be sent to these destinations

aws_1
Webhook Id

After detecting an event:

The webhook Endpoint supplied to the above endpoint is that of an AWS lightsail container, which is basically a container application (a python flask app in this case) that is run on the cloud using Docker Images.



flask-service
Container service
Micro (1 GB RAM, 0.25 vCPUs) ×1 node
Mumbai

Status: Running

Public domain: flask-service.5emqdmvmdhp3q.ap-south-1.cs.amazonlightsail.com
How do I use my domain with my container service?
Private domain: flask-service.service.local

Current deployment
Your deployment is the set of containers currently running on your container service.
Learn more about deployments
Modify your deployment

Deployed: July 28, 2023 - 6:55 PM

CONTAINERS
Public endpoint
flask
Image: :flask-service.flask-container.31
Open log

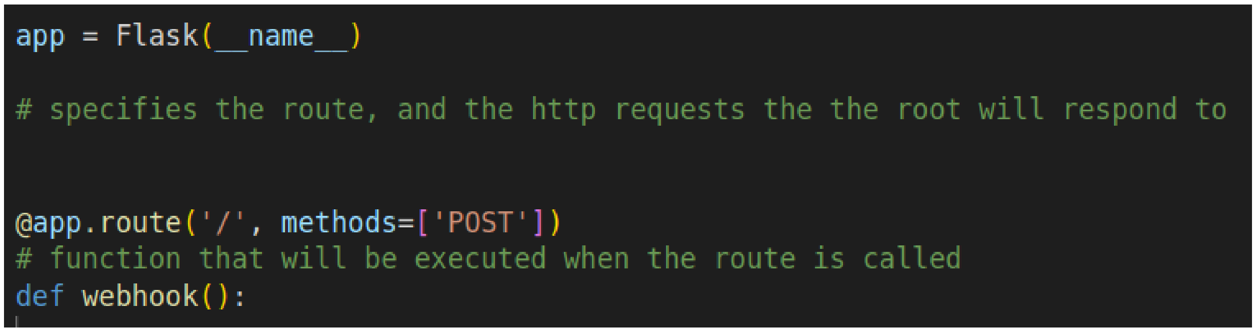
PUBLIC ENDPOINT
flask

Port	Protocol
5000	HTTP

Health check path
/
More details

Again, there will be specific documentation regarding setting up the AWS container and dockerizing the flask app.

The python code running on the container executes a function everytime it receives a POST request (which occurs every time Tenderly sends an alert)



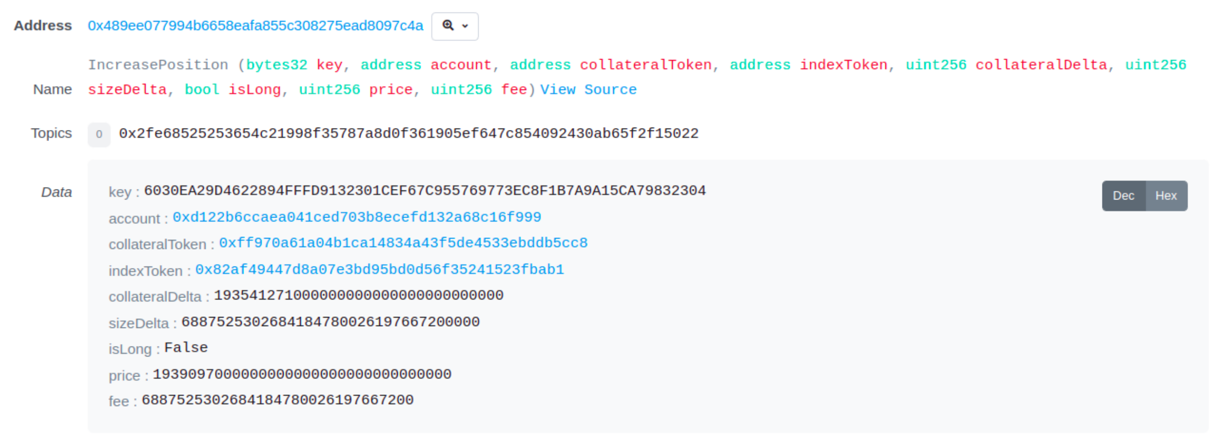
```python
app = Flask(__name__)

# specifies the route, and the http requests the the root will respond to


@app.route('/', methods=['POST'])
# function that will be executed when the route is called
def webhook():
```

Basically, when Tenderly detects that the specific event is emitted, it sends an alert to an AWS container, which is received by the flask app running on it, which leads to a python function getting executed.

Date Extraction Code:

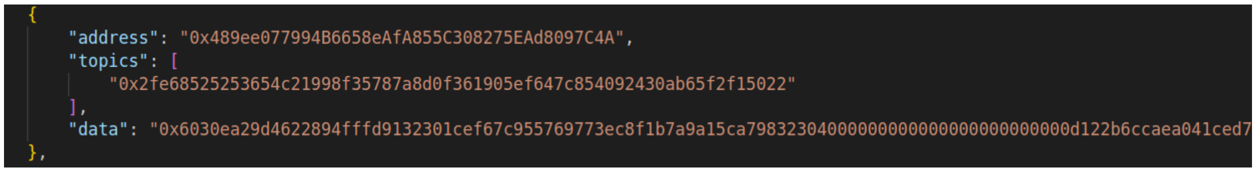
The function so triggered receives data in the form of an extensive raw JSON file, most of whose content is in hexadecimal.

For example, the 'IncreasePosition' event of the transaction with hash '0xebc4a0e1c6ced934cc4d4e98b77ddd317033bafdd22a6c74035f54865630422b8', so represented on Arbiscan



```
Address   0x489ee077994b6658eafa855c308275ead8097c4a

Name      IncreasePosition (bytes32 key, address account, address collateralToken, address indexToken, uint256 collateralDelta, uint256
          sizeDelta, bool isLong, uint256 price, uint256 fee) View Source

Topics    0  0x2fe68525253654c21998f35787a8d0f361905ef647c854092430ab65f2f15022

Data      key : 6030EA29D4622894FFFD9132301CEF67C955769773EC8F1B7A9A15CA79832304
          account : 0xd122b6ccaea041ced703b8ecefd132a68c16f999
          collateralToken : 0xff970a61a04b1ca14834a43f5de4533ebddb5cc8
          indexToken : 0x82af49447d8a07e3bd95bd0d56f35241523fbab1
          collateralDelta : 1935412710000000000000000000000
          sizeDelta : 6887525302684184780026197667200000
          isLong : False
          price : 1939097000000000000000000000000000
          fee : 6887525302684184780026197667200
```

will be found as a raw long in the received json file.



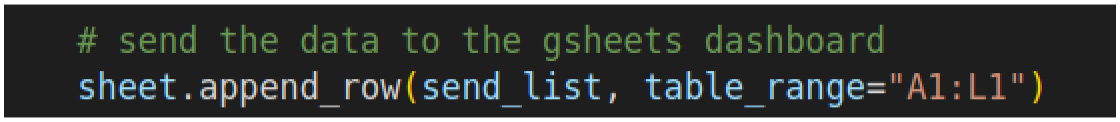
```
{
    "address": "0x489ee077994B6658eAfA855C308275EAd8097C4A",
    "topics": [
        "0x2fe68525253654c21998f35787a8d0f361905ef647c854092430ab65f2f15022"
    ],
    "data": "0x6030ea29d4622894fffd9132301cef67c955769773ec8f1b7a9a15ca79832304000000000000000000000000d122b6ccaea041ced7
},
```

Detecting certain events and extracting their specific parameters is the work of an extensive dashbaord code, which will again have separate documentation.

The codes used are 'open_trades_notif_extraction_dashboard.py' and 'closing_trades_notif_extraction_dashboard.py' in the 'Auto-Trading' Folder of the Auto-Trading Repo.

Dashboard recording relevant Trades:

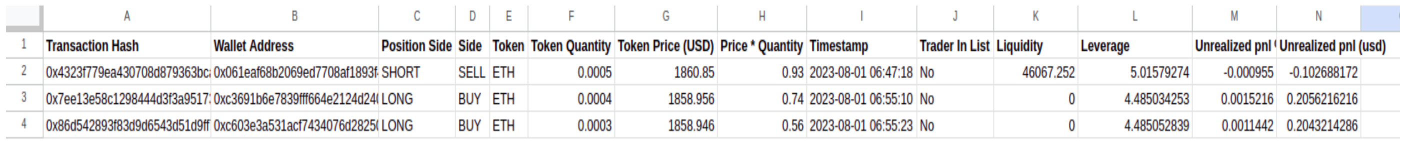
The above flask app eventually sends the extracted params to a Google Sheets Dashboard.



```
    # send the data to the gsheets dashboard
    sheet.append_row(send_list, table_range="A1:L1")
```

There exists a separate dashboard for recording open Trades, and another for recording Closing Trades.

Following are the attributes of the current 'Open Trades Dashboard' for instance.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Transaction Hash	Wallet Address	Position Side	Side	Token	Token Quantity	Token Price (USD)	Price * Quantity	Timestamp	Trader In List	Liquidity	Leverage	Unrealized pnl (	Unrealized pnl (usd)
2	0x4323f779ea430708d879363bca	0x061eaf68b2069ed7708af1893f	SHORT	SELL	ETH	0.0005	1860.85	0.93	2023-08-01 06:47:18	No	46067.252	5.01579274	-0.000955	-0.102688172
3	0x7ee13e58c1298444d3f3a9517	0xc3691b6e7839fff664e2124d24	LONG	BUY	ETH	0.0004	1858.956	0.74	2023-08-01 06:55:10	No	0	4.485034253	0.0015216	0.2056216216
4	0x86d542893f83d9d6543d51d9ff	0xc603e3a531acf7434076d2825	LONG	BUY	ETH	0.0003	1858.946	0.56	2023-08-01 06:55:23	No	0	4.485052839	0.0011442	0.2043214286

The dashboards are updated in real-time, as and when new data comes in.

Following are the links to the dashboards:

For Opening Trades:
https://docs.google.com/spreadsheets/d/1c7wHPZ9CgaXrVtL24PduIl8xdb4yzx8BlWj5Zh6sN0M/edit?usp=sharing

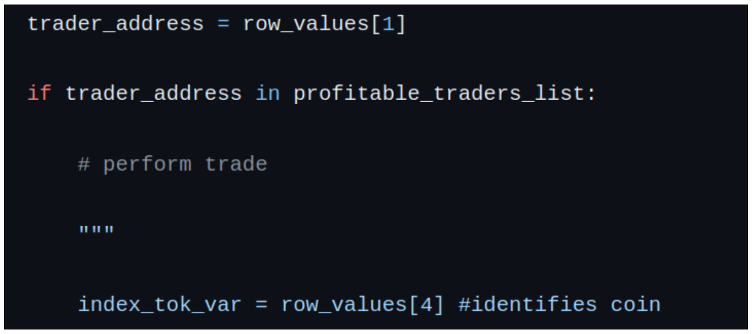
For Closing Trades:
https://docs.google.com/spreadsheets/d/1B2nyzH_EEfpTqjpCqYdltZCNL9fVZpvvE3hbpz6vPOs/edit?usp=sharing

Executing the Trades:

Now there exists a separate python code (running continuously on an AWS lightsail server instance) which detects when a new row has been added to the Google Sheet (indicating that a new relevant trade has been made), which then determines whether we must execute a corresponding transaction. If yes, it does so on our given binance account.

A base code to do this is present as 'trading_open.py' and 'trading_closed.py' in the Auto-Trading repo.

To determine whether we ought to consider the trade, it first checks whether the trader who executed the trade is part of a given 'profitable_trader_list'.

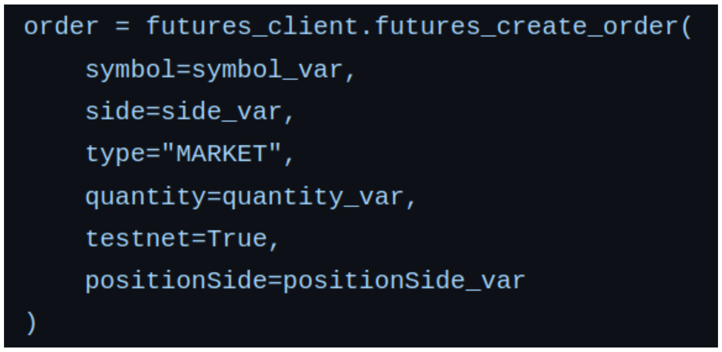


```
trader_address = row_values[1]

if trader_address in profitable_traders_list:

    # perform trade

    """

    index_tok_var = row_values[4] #identifies coin
```

To execute a trade on the binance account if required, the python code makes use of Binance's API.

One must possess the required API key and Secret Key to make such a transaction, and the device from which the request comes from must have its IP first individually whitelisted by the binance account.

The following snippet for example opens a position in 'Binance Futures'.



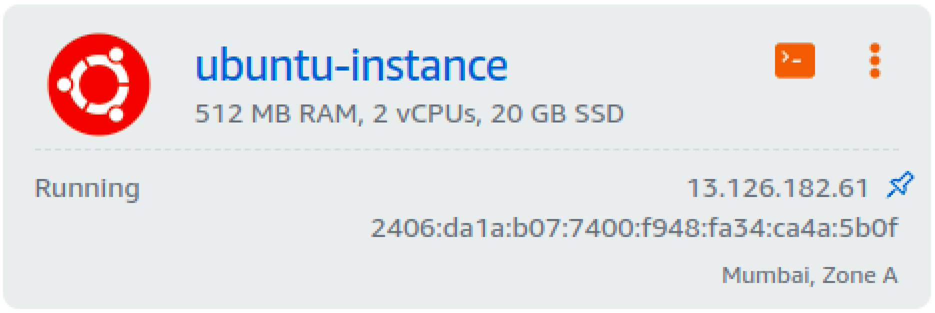
```
order = futures_client.futures_create_order(
    symbol=symbol_var,
    side=side_var,
    type="MARKET",
    quantity=quantity_var,
    testnet=True,
    positionSide=positionSide_var
)
```

One may read the API documentation for further information on future required functionalities.
https://python-binance.readthedocs.io/en/latest/

Now the above code must be continuously executed to detect real-time trades. At the same time, it must also be executed on a server/container that has a static ip address that never changes, else it would lose all the trading permissions that it obtained by the manual whitelisting of its IP address in the binance account.

The lightsail container-service used previously does not support static IP addresses. To do this, an AWS lightsail instance is employed.

A lightsail instance is essentially a virtual private server hosted on the cloud, based on pre-configured operating systems/applications.
In this case, it is based on 'Ubuntu 20.04 LTS'.



ubuntu-instance
512 MB RAM, 2 vCPUs, 20 GB SSD
Running
13.126.182.61
2406:da1a:b07:7400:f948:fa34:ca4a:5b0f
Mumbai, Zone A

There will be separate documentation regarding the AWS structure and hosting.

—---

The document thus explains the basic workflow of the entire automatic-trading system, while also providing a brief outline of each of its stages.