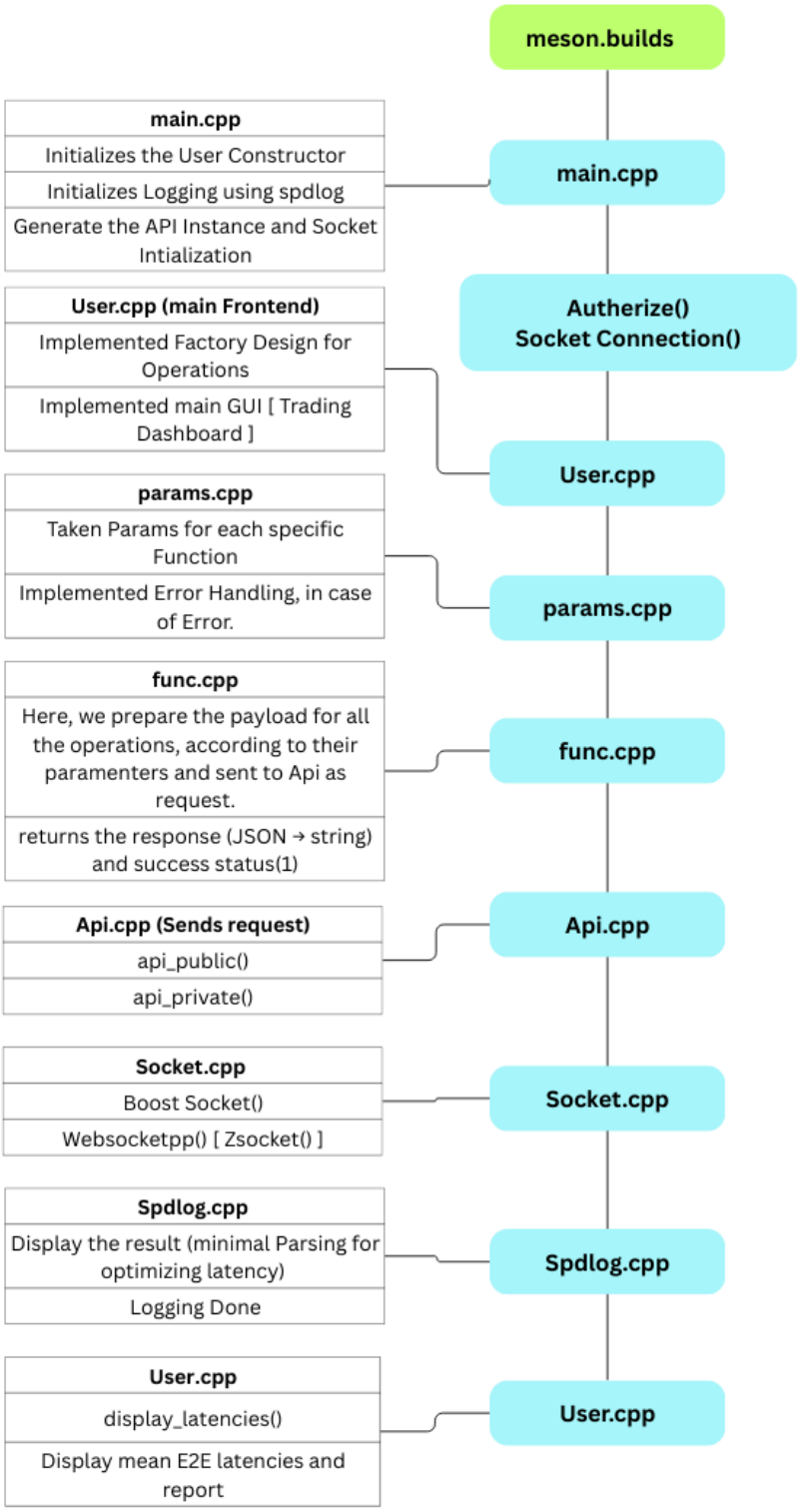
Documentation Report

Structure:-



Code:-

Class:: User

```
class User
{

public:
   User();
   void Start();

   std::pair<int, std::string> place_order(const std::string &, double, int);
   std::pair<int, std::string> cancel_order(const std::string &);
   std::pair<int, std::string> modify_order(const std::string &, double, int);
   std::pair<int, std::string> get_orderbook(const std::string &, int);
   std::pair<int, std::string> view_position(const std::string &);
   std::pair<int, std::string> get_openorders(const std::string &);
   std::pair<int, std::string> get_marketdata(const std::string &, int);

   int display_response(const std::string &resp);
   int process_cancel_order(const function<pair<int, string>(string)> &action);
   int process_place_order(const function<pair<int, string>(string, double, int)> &action);
   int process_modify_order(const function<pair<int, string>(string, double, int)> &action);
   int process_order_book(const function<pair<int, string>(string, int)> &action);
   int process_view_position(const function<pair<int, string>(string)> &action);
   int process_open_orders(const function<pair<int, string>(string)> &action);
   int process_market_data(const function<pair<int, string>(string, int)> &action);

   void display_mean_latencies();


private:
   Api *m_api;
   std::vector<double> order_latencies;
   std::vector<double> market_data_latencies;
   std::vector<double> websocket_latencies;
   std::vector<double> trading_loop_latencies;
};
```

Class:: Api

```
class Api
{
```

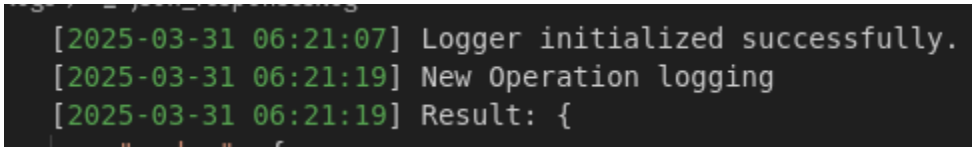
```
public:
    // Constructor
    Api();
    Api(const std::string &client_id, const std::string &client_secret);

    // Destructor
    ~Api();

    // Methods
    [[nodiscard]] std::pair<int, std::string> api_public(const std::string &msg);
    [[nodiscard]] std::pair<int, std::string> api_private(const std::string &msg);

    [[nodiscard]] int Authenticate();


private:
    Socket *m_socket;
    std::string m_client_id;
    std::string m_client_secret;

    std::string generate_auth_msg() const;
    void initialize_socket(); // Add this declaration
};
```

Class:: Socket

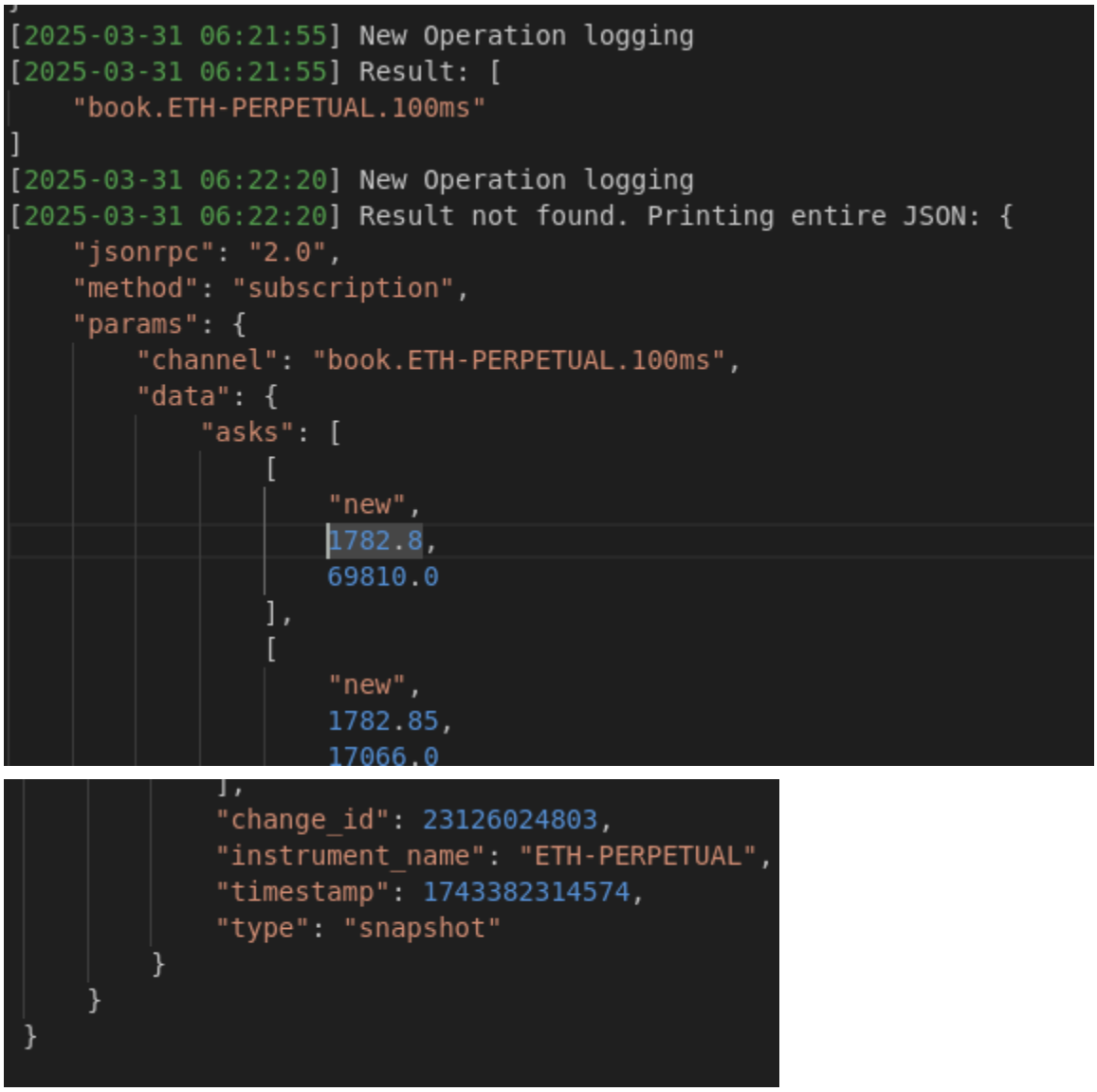
```
class Socket {
public:
    virtual ~Socket();
    virtual void switch_to_ws() = 0;
    [[nodiscard]] virtual std::pair<int, std::string> ws_request(const std::string& msg) = 0;

protected:
    const std::string host = "test.deribit.com";
    const std::string port = "443";
};
```

Features:-

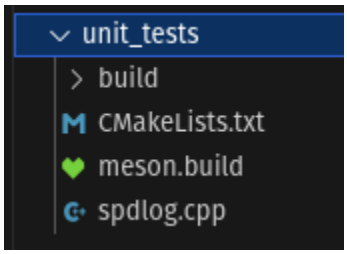
1. Uses Spdlog for proper and timed logging.



2. Market data Streaming does not override the main program.

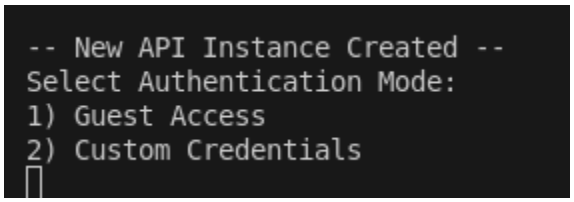


3. Created Unit test for testing network and logging operations.

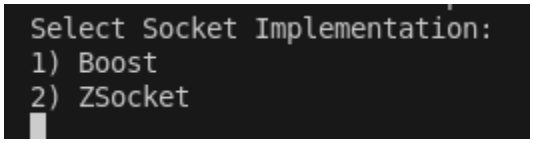


4. Implemented Boost Socket and Zsocketpp and performed benchmarking for both.
5. Used Meson Build for higher performance and faster build times.

6. Implemented Guest Mode and Custom Credentials for User.



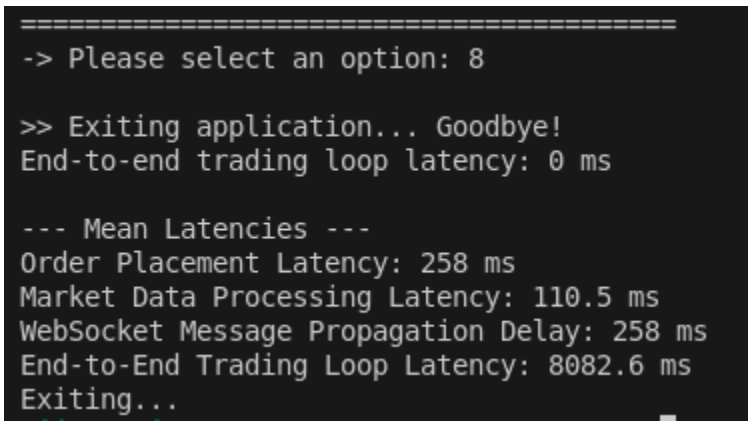
7. Implemented choice based Websocket selection.



8. Used Class Based Code for better modularity and implemented Factory Pattern.

```
int status = process_place_order([this](const std::string &inst, double price, int quant)
                                 { return place_order(inst, price, quant); });
std::cout << (status ? "!! Failed to place the order.\n" : ">> Order placed successfully!\n");
break;
```

9. Implemented codes for multiple Latency Records at various steps for easily checking the performance at a niche level.



ZSOCKET:-

```
-> Please select an option: 1
Instrument (e.g., BTC-PERPETUAL, ETH-PERPETUAL): BTC-PERPETUAL
Price: 50
Quantity: 50
WebSocket message propagation delay: 217 ms
Order placement latency: 217 ms
Response:
```

```
>> Order placed successfully!
End-to-end trading loop latency: 7360 ms
```

BOOST:-

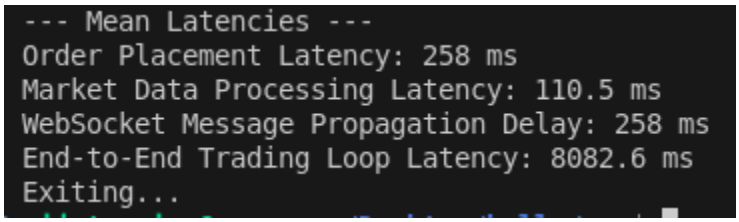
```
-> Please select an option: 1
Instrument (e.g., BTC-PERPETUAL, ETH-PERPETUAL): BTC-PERPETUAL
Price: 50
Quantity: 50
WebSocket message propagation delay: 456 ms
Order placement latency: 456 ms
Response:
```

```
>> Order placed successfully!
End-to-end trading loop latency: 8847 ms
```

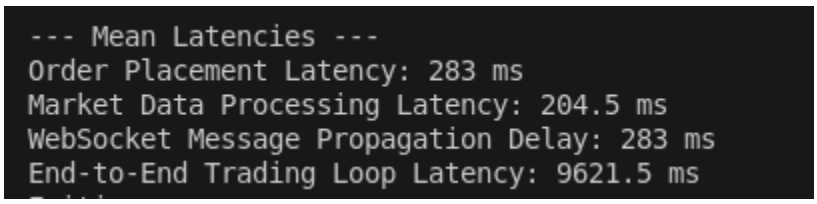
Performance Analysis and Optimization

Latency Benchmarking:-

For Zsocket:-

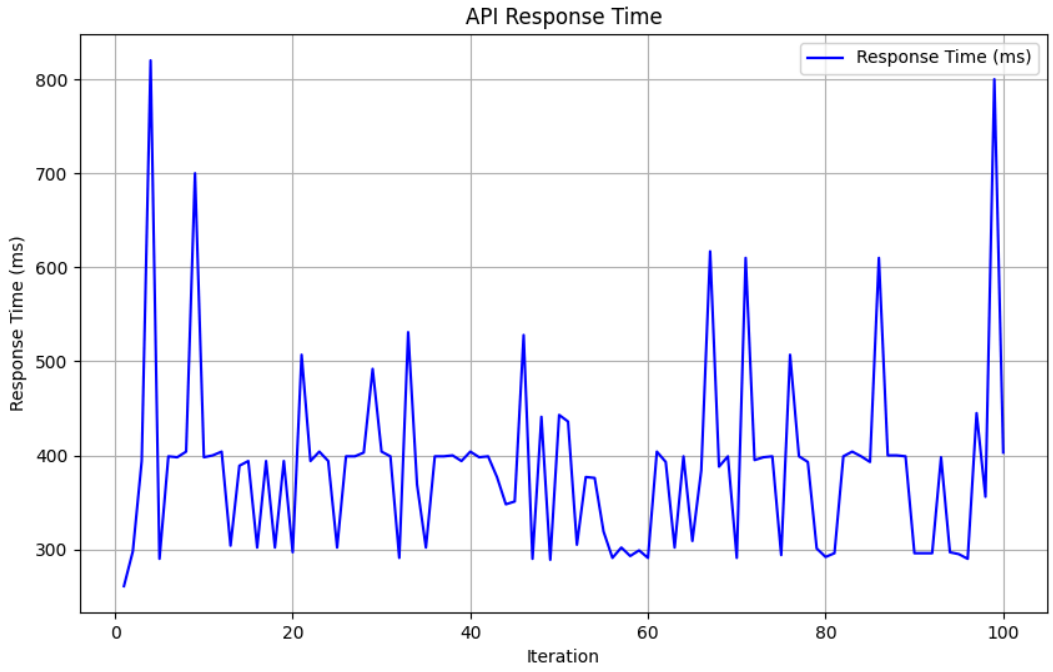


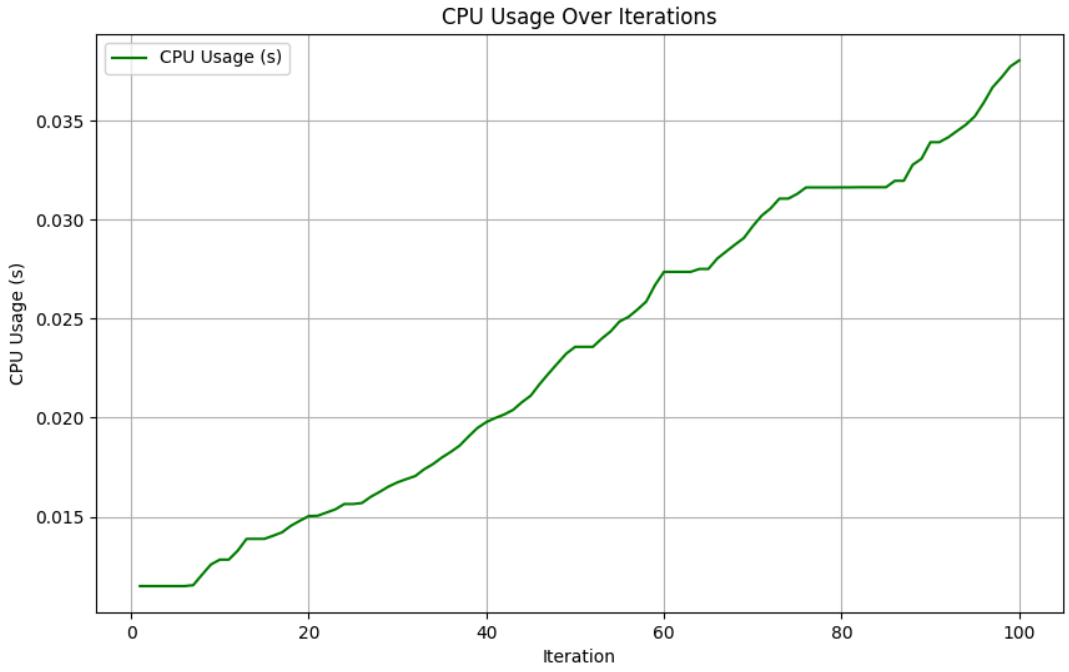
For Boost:-

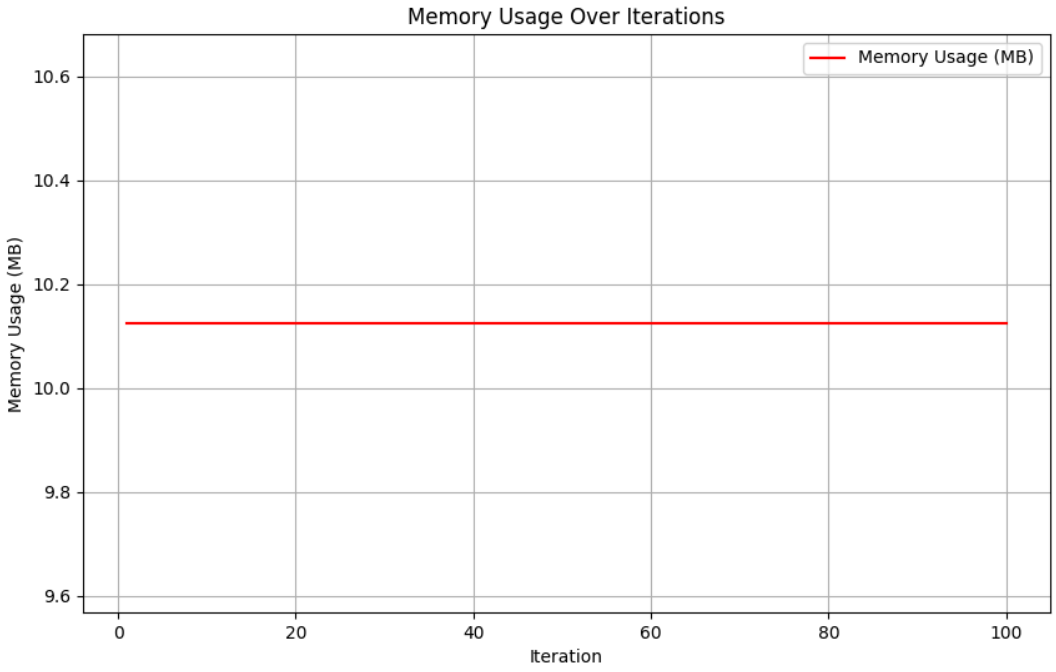


Optimization Requirements:-

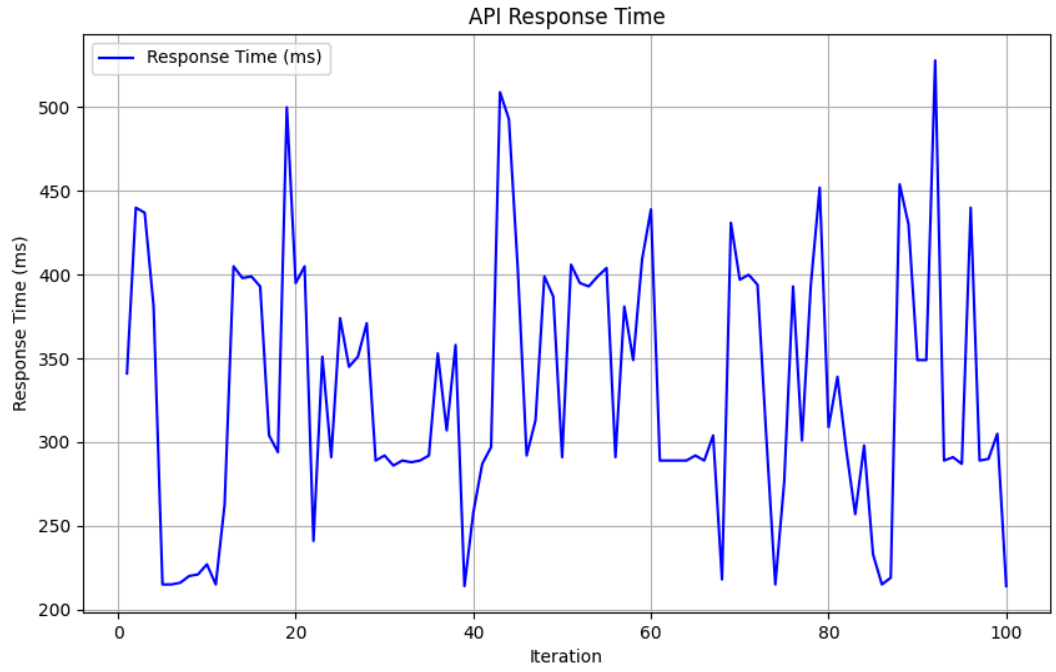
For Boost:-

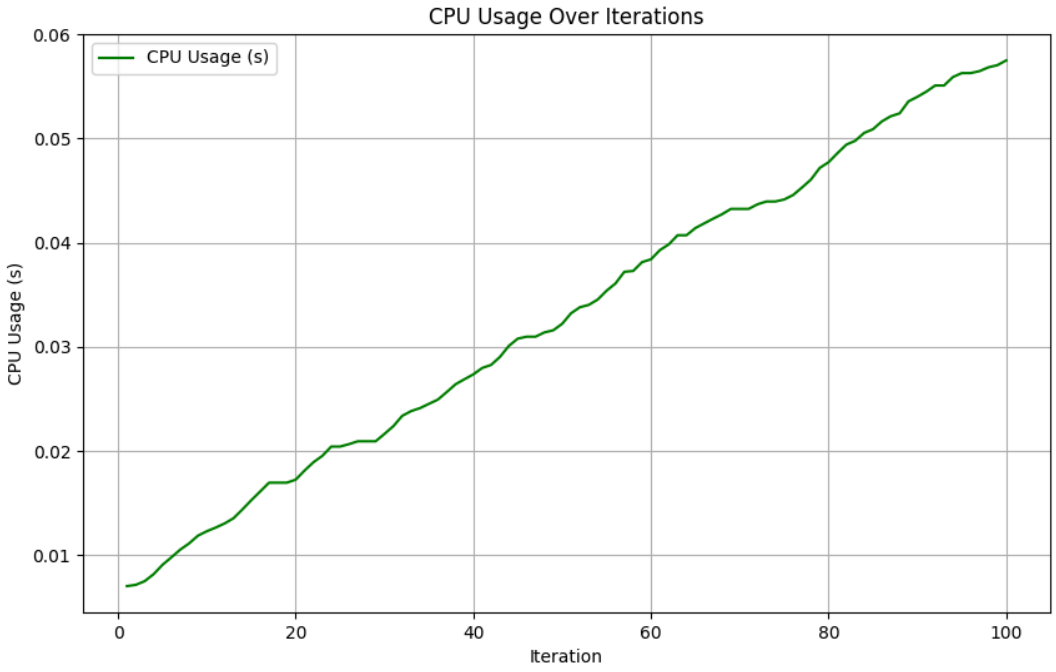

API Response Time
Response Time (ms)
Response Time (ms)
800
700
600
500
400
300
0
20
40
60
80
100
Iteration

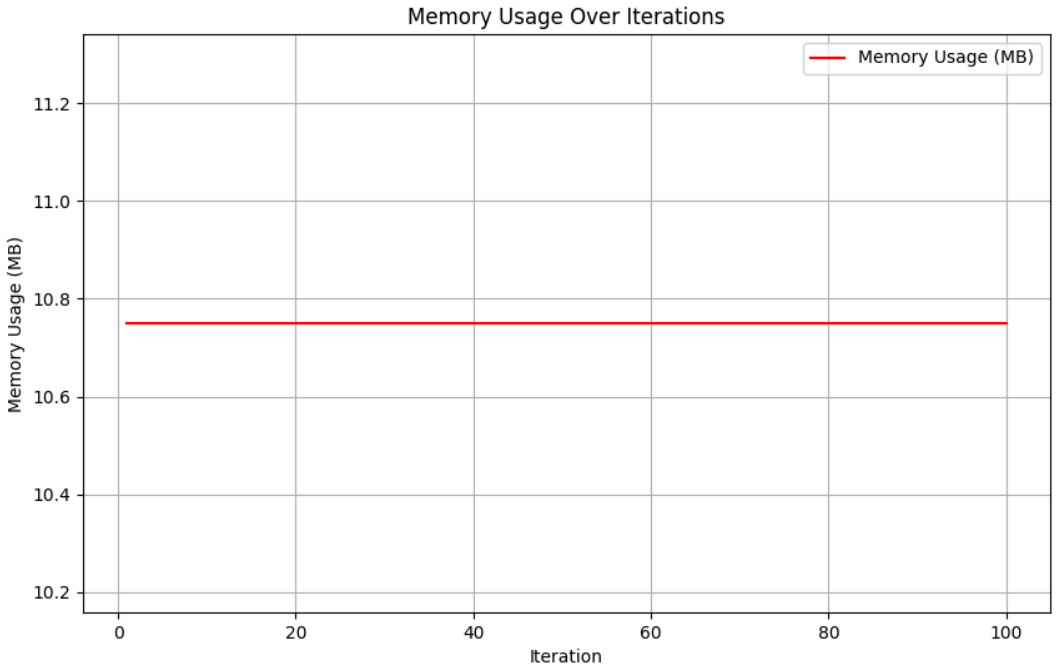

CPU Usage Over Iterations
CPU Usage (s)
CPU Usage (s)
0.035
0.030
0.025
0.020
0.015
0
20
40
60
80
100
Iteration


Memory Usage Over Iterations
Memory Usage (MB)
Memory Usage (MB)
10.6
10.4
10.2
10.0
9.8
9.6
0
20
40
60
80
100
Iteration

For Zsocket:-


API Response Time
Response Time (ms)
Response Time (ms)
500
450
400
350
300
250
200
0
20
40
60
80
100
Iteration





Conclusions:-

1. **Zsockets take relatively less API response time than Boost Socket.**

2. **BUT Boost Socket uses less Memory and CPU then Zsocket.**
3. **So, it's a tradeoff between CPU Usage and Response Time.**

Data Structure:-

For **Multi-Users**, we can implement **unordered_map<int,unordered_set<int>>** for storing the Market Streaming Subscriber Data, to keep record of all the subscriptions according to each different User.

1. Unordered Set ensures that there are no duplicate values of Channel for one user and uses less time for insert time and removal time for a channel for a particular user.
2. Unordered Map ensures that there are less time for inserting and removing channel for a particular User.
3. Aside for this, we can also use **unordered_map<unordered_map<channel_id(string), bool>>.**

Documentation Requirements for Bonus Section
1. Detailed analysis of bottlenecks identified
2. Benchmarking methodology explanation
3. Before/after performance metrics
4. Justification for optimization choices
5. Discussion of potential further improvements

Bottlenecks Identified:-

1. **Market Data Streaming:-**
   Issue:- When we Stream Market Data, due to incoming messages continuously, the system often crashes.
   Approach:-
   1. Stream Market data in another terminal, using two sockets. First one for Streaming Market Data every 100ms. So, that it does not override the main function.
   2. Stream Market Data in the the same terminal but using multithreading, so that it will not override the main function.

   Conclusion:-
   Implemented the first one halfway, so that it does not override the main function.

Needs to be implemented:- The market data streaming in another terminal can be implemented. The main challenge is to implement two sockets for different message and data streaming.

Benchmarking Methodology :-

1. Used Matplotlib and Python3 to generate the graphs for Zsocket and Boost.
2. Used benchmarking tools like chrono, to measure latency and create a .csv file [in the Benchmarking Folder].
3. Implemented Unit Tests to test Spdlog for proper and timed logging.

Further Benchmarkings:-

1. Using perf -stat and record to see the operations going in the background while running the application.
2. Using gtest (google testing) to benchmark for a particular operations and mock connection for websockets.
3. Producing FlameGraphs to properly benchmark the resources used by our function, to properly benchmark the application.

Json Logging Response (for one whole Instance)