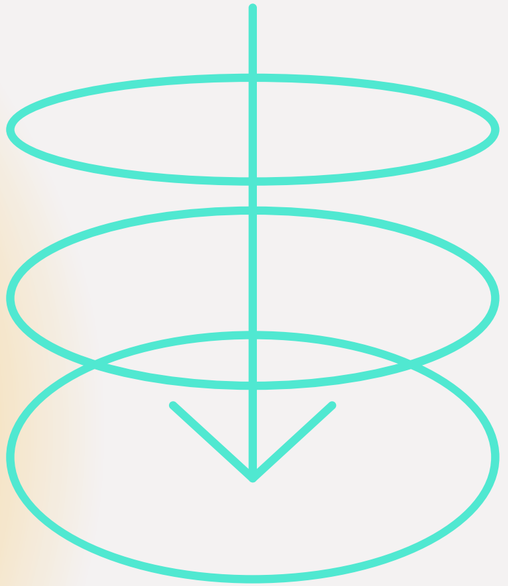
# Team Name: Enigma

# Visualising Options Data and Calculating Key Metrics




**Members:**

Hetvi Gudka (16010120010)
Kaushik Iyer (16010120017)
Akhil Nagar (16010120031)
Avish Rodrigues (16010420060)

# Introduction

**Problem Statement:**

- The problem statement for the hackathon is to build an options chain tool.
- The tool should process a market data stream received over TCP/IP and calculate implied volatility (IV) based on the Black-Scholes formula.
- The goal is to display the options chain as a real-time webpage, allowing users to select different underlyings and expires.
- The options chain should highlight "in the money" and "out of money" options.
- The tool should update and recalculate the data on the screen dynamically without reloading the browser.

# Technologies and Concepts Used

## Apache Kafka

- Parallel processing utilized multiple threads to execute tasks simultaneously, improving performance and efficiency.
- Data processing tasks were divided into segments, each assigned to a separate thread for simultaneous execution.
- Implied volatility calculations were parallelized across multiple threads, reducing computation time.
- Efficient task scheduling and load balancing ensured even distribution of workload.
- Parallel processing optimized resource utilization and provided real-time updates for the options chain UI.
- Enhanced performance, faster data processing, and improved user experience.

# Technologies  and Concepts Used

## Socket Programming:

- Socket programming is a programming paradigm that enables communication between two computers over a network using sockets. Sockets provide a mechanism for processes to establish network connections and exchange data. In the context of this project, socket programming was used to receive a market data stream over TCP/IP.
- The participants needed to implement a client application that connects to this server using sockets. The client application would establish a TCP/IP connection with the server and listen for incoming data.
- By utilizing socket programming, the client application could receive the market data packets sent by the server in real-time. These packets contained various fields such as trading symbol, timestamp, last traded price, volume, bid/ask prices, and more.
- Socket programming facilitated the continuous and dynamic exchange of market data between the server and the client application. It allowed for real-time updates of the options chain without the need to reload the browser, ensuring that the displayed information remained current and relevant to the changing market conditions.

# Technologies and Concepts Used

## Multithreading:

- Multithreading was employed to enable concurrent execution of tasks, enhancing the performance and responsiveness of the application.
- Synchronization mechanisms, such as locks and mutexes, were utilized to safeguard shared data structures and prevent race conditions when multiple threads accessed and modified data simultaneously.
- Communication mechanisms, such as message queues or shared variables, facilitated efficient and safe data exchange between the processing thread and the UI update thread.
- By leveraging multithreading, the application achieved efficient data processing and real-time UI updates, providing traders with a seamless and responsive experience while interacting with the options chain tool.
- Overall, the utilization of multithreading improved the system's performance, responsiveness, and user experience, enabling traders to make informed decisions based on up-to-date market data.

# Technologies and Concepts Used

## Parallel Processing

- Parallel processing utilized multiple threads to execute tasks simultaneously, improving performance and efficiency.
- Data processing tasks were divided into segments, each assigned to a separate thread for simultaneous execution.
- Implied volatility calculations were parallelized across multiple threads, reducing computation time.
- Efficient task scheduling and load balancing ensured even distribution of workload.
- Parallel processing optimized resource utilization and provided real-time updates for the options chain UI.
- Enhanced performance, faster data processing, and improved user experience.

# Data Processing and Calculations

- Implemented a robust data processing pipeline to handle the market data stream effectively.
- Utilized various algorithms and formulas to perform calculations and derive key metrics.
- Market data packets were received and parsed in real-time from the Kafka message stream.
- Extracted relevant information such as trading symbol, last traded price, volume, bid/ask prices, and open interest.
- Implemented implied volatility (IV) calculation using the Black Scholes formula.
- Leveraged the underlying price data from the market stream for accurate IV calculation.
- Considered risk-free interest rate of 5% for IV computation.
- Accurately calculated the time to maturity (TTM) based on the expiry date and time assumptions.
- Highlighted "in the money" and "out of money" options in the options chain UI.
- Continuously refreshed and recalculated the data and metrics as the market data stream updated, providing real-time insights.
- Ensured efficient data processing and optimized resource utilization through parallel processing and multithreading.
- The processed data and calculated metrics were seamlessly integrated into the options chain screen, providing users with up-to-date and accurate information.

# Realtime Updates

- Achieving real-time updates was a crucial aspect of the options chain tool.
- Implemented a reactive architecture that enabled real-time data processing and display.
- Utilized web sockets to establish a persistent connection between the server and the client.
- Enabled bi-directional communication, allowing the server to push updates to the client instantly.
- Leveraged event-driven programming to handle real-time updates efficiently.
- As the market data stream changed, the server processed the updates and pushed them to the connected clients.
- Implemented efficient data diffing algorithms to identify changes in the market data and transmit only the updated information.
- Applied asynchronous programming techniques to handle concurrent data processing and update delivery.
- Leveraged modern JavaScript frameworks like React or Angular on the client-side to efficiently handle and render real-time updates.
- The client's UI was automatically updated without requiring manual page reloads or user interactions.
- Provided a smooth and responsive user experience by minimizing data latency and ensuring near-instantaneous updates.

# Advantages of the chosen approach

- **Real-time data processing:** The script enables real-time data processing by continuously receiving packets from the server and processing them as they arrive. This allows for immediate analysis and actions based on the received data.
- **Scalability:** The script uses threading to handle packet processing in parallel. This can improve performance and scalability by processing multiple packets concurrently, taking advantage of multi-core processors.
- **Streaming data integration:** By utilizing the Confluent Kafka library, the script seamlessly integrates with Kafka, a popular distributed streaming platform. It enables the processed data to be efficiently published to Kafka topics, making it available for further processing, analysis, or consumption by other systems.
- **Customizable processing logic:** The script provides flexibility to define custom processing logic for each received packet. It unpacks the fields from the packet and performs calculations, such as calculating implied volatility. This allows for tailoring the processing based on specific requirements or business logic.

# Advantages of the chosen approach

- **Error handling:** The script includes error handling mechanisms to handle potential exceptions or errors during socket communication, packet processing, and socket closure. This helps ensure the script's robustness and stability.

- **Modularity and maintainability:** The script separates the packet processing logic into functions, which enhances code modularity and maintainability. It allows for easier understanding, testing, and modification of specific components.

- **Integration with external systems:** The script demonstrates integration with external systems, such as a server sending packets and Kafka for data streaming. This can be extended to integrate with other systems, databases, or analytics frameworks based on specific needs.

# Implementation

- **Establish Socket Connection:** The Python script initiates a socket connection with a server to receive a continuous stream of options data. This connection allows for efficient and real-time data retrieval.

- **Process Options Data:** The received options data is processed within the script, involving complex calculations to determine the implied volatility (IV) associated with each option. This step utilizes mathematical models and statistical techniques to derive meaningful insights from the data.

- **Queue Data with Apache Kafka:** The processed IV values are stored in a queue implemented through Apache Kafka. Kafka's reliable and scalable distributed streaming platform ensures efficient data handling and smooth data flow. The queue acts as a temporary repository for the computed IV values before further processing.

- **Integrate Streamlit for data visualization:** Incorporate the Streamlit library into the Python script to provide a user-friendly interface for displaying the options data and IV values.

# Implementation

- **Integration with Confluent Kafka Python Library:** The Confluent Kafka Python library is utilized to seamlessly integrate the script with Apache Kafka. This library provides high-level abstractions and functionalities to interact with Kafka, simplifying the implementation of data transmission to Kafka topics.

- **Update the table upon queue updates:** Continuously monitor the Kafka queue for updates. Whenever a new IV value is enqueued, retrieve it from the queue and update the corresponding table in Streamlit. This ensures that the displayed table remains up to date with the latest IV values.

- **Continuously run the script:** Deploy the Python script in a persistent manner to continuously receive, process, and store the options data. This ensures that the system operates in real-time, providing the most recent IV values and keeping the Streamlit table continuously updated for users.

# Challenges Faced

1. **Network connectivity:** Establishing and maintaining a stable network connection between the client and server is crucial. Issues such as network outages, intermittent connectivity, or high latency can impact the script's ability to receive packets in a timely manner or lead to connection failures.
2. **Packet integrity:** The script assumes that each packet received has a fixed size of 130 bytes. If the actual packet size differs or if there are errors in the packet structure, it may cause issues during packet processing, such as failed unpacking or incorrect data extraction.
3. **Data synchronization:** The script is designed to receive packets in real-time. If there is a delay or gap in packet reception, it could lead to data inconsistencies or missing data points. It's important to ensure proper synchronization between the server sending the packets and the script receiving them.
4. **Error handling and fault tolerance:** The script includes some error handling mechanisms, but it may be necessary to enhance the error handling and implement fault tolerance measures to gracefully handle exceptions, socket timeouts, or unexpected issues. This is crucial to maintain the script's stability and prevent data loss.
5. **Scalability and performance:** Depending on the packet rate and processing complexity, the script's performance and scalability should be considered. If the packet rate is high or the processing logic is computationally intensive, it may be necessary to optimize the code, consider multi-threading, or distribute the processing across multiple nodes to handle the load efficiently.

# Future Scope

1. **Modularize the code:** Break down the code into smaller, reusable functions or classes. This promotes code reusability, maintainability, and allows for easier testing and debugging. Separate concerns such as packet processing, Kafka integration, error handling, and data validation into separate modules.
2. **Implement logging and error handling:** Enhance the error handling mechanisms by implementing comprehensive logging. Log important events, errors, and exceptions to aid in troubleshooting and debugging. Consider using a logging library that supports different log levels and allows configuration options for log output.
3. **Use configuration files:** Externalize configuration parameters such as Kafka bootstrap servers, topics, packet sizes, and timeouts into a configuration file. This allows for easier configuration management without modifying the code itself. Consider using a library like configparser or YAML files for configuration.
4. **Implement unit tests:** Write unit tests for critical functions and components of the codebase. Unit tests help ensure the correctness of the code, identify potential issues early on, and provide confidence when making changes or refactoring.
5. **Optimize packet processing:** Analyze the packet processing logic and explore potential optimizations. Look for opportunities to reduce unnecessary computations, improve algorithm efficiency, or utilize multi-threading or asynchronous processing techniques to handle high packet rates.

# References


1. "Options: Calls and Puts." Investopedia. Available at: https://www.investopedia.com/terms/o/option.asp
2. "Option (Finance)." Wikipedia. Available at: https://en.wikipedia.org/wiki/Option_(finance)
3. "Black–Scholes model." Wikipedia. Available at: https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model
4. "Apache Kafka." Apache Kafka Documentation. Available at: https://kafka.apache.org/
5. "WebSocket." MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
6. "React." React Documentation. Available at: https://reactjs.org/
7. "Angular." Angular Documentation. Available at: https://angular.io/