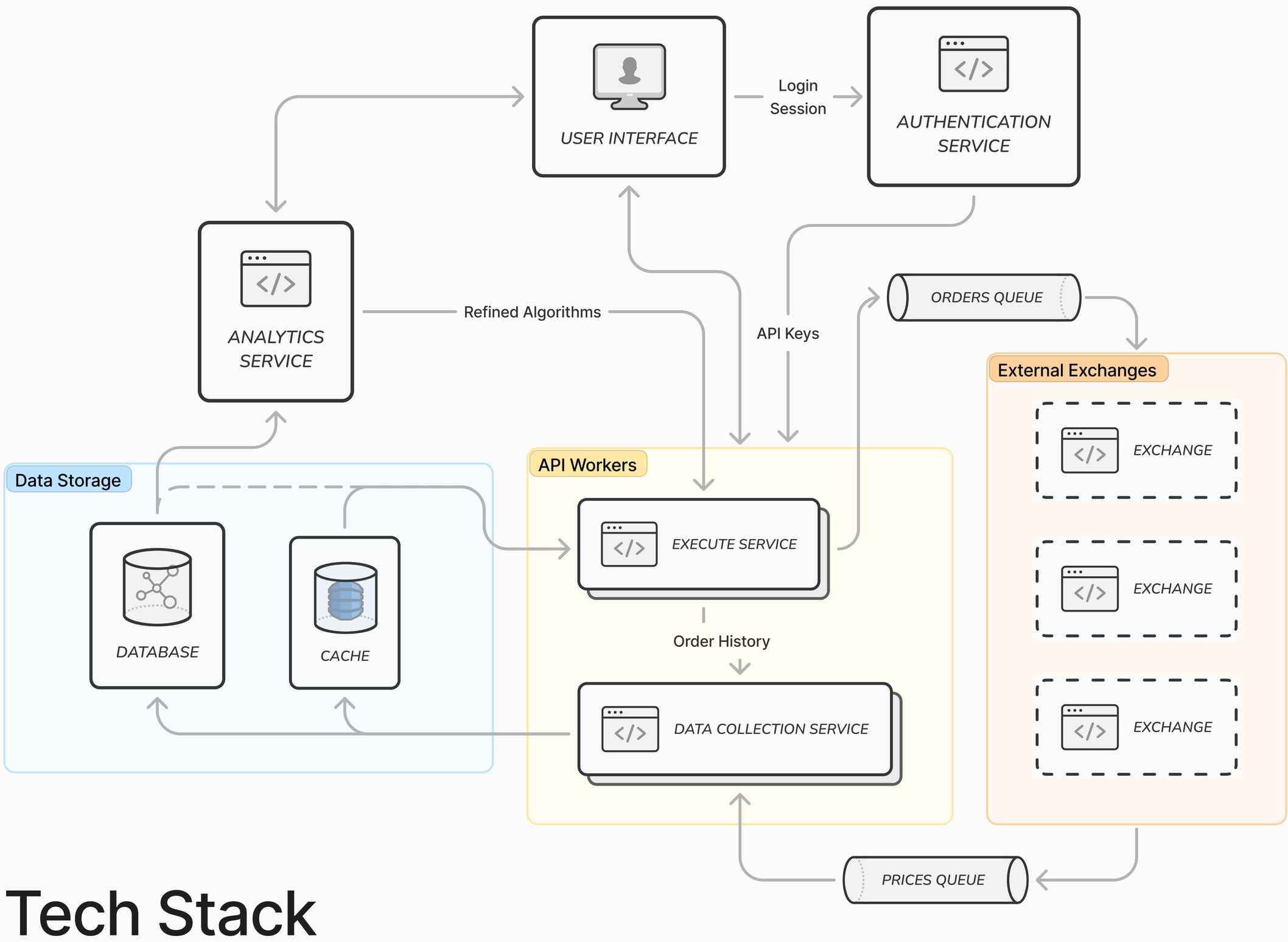

Tech Stack

User Interface

- Enable user to set trading parameters (e.g. risk tolerance, trade frequency, profit level) and execute automated trades
- Reporting of historical trades, performance and trade strategy recommendations

Authentication Service

- After user is authenticated, create a user session and authenticate API endpoints with valid keys

Analytics Service

- Performs back-testing of the trading algorithm on historical data to iterate on trading strategies and provide the most optimum ones

Database

- Store historic price and order-execution data for back-testing
- Use a time-series database to maintain performance with large data volumes

Cache

- Store short-term price data for quick retrieval

Message Queues

- Support asynchronous endpoint calls and offer data persistence during system failures

Data Collection Service

- Consumes pricing/order data and forwards to data storage
- Multiple workers retrieve price updates from each exchange via web sockets to minimize latency

Execute Service

- Enacts decision logic for performing buy/sell orders that take advantage of arbitrage opportunities
- Takes into account user-supplied trading parameters (risk tolerance, limit/stop orders, trade frequency, minimum profit...)
- Each buy/sell order is performed by a separate worker to minimize latency

Design Choices

Optimize for low latency

- Price quotes from each exchange must be as up-to-date as possible in order to maximize the number of arbitrage opportunities than can be capitalized upon. Execution orders must also be completed as soon as possible to limit slippage. Hence, web socket connections are used to maintain bi-directional channels for real-time data transfer between the exchanges and data services
- Short-term pricing data is stored in a cache, enabling faster data access to capitalize on transient arbitrage opportunities
- Multiple workers for data collection and order execution for asynchronous calls and lower latency

Fault tolerance

- Message queues used to persist pricing/order data in the event of failures (e.g. network, service)
- Database used as a backup in the case of cache/memory failure

Security

- API authentication via keys to protect API rate/call limits and prevent unauthorized activity (i.e. execution orders)

General trade-offs

- Complexity was introduced in the above measures to lower latency and increase system robustness, leading to slower development and more challenging maintenance
- More nuanced trading strategies are provided in the analytics and user-supplied parameters to maximize profitability in unpredictable market conditions. However, this is more complex and slower to implement than the naive algorithm of simply comparing price difference between exchanges
- To demonstrate a simple proof of concept, the system above is designed for a single user and not currently scalable to multiple users