Performance Optimization Approaches

Ensuring a responsive, real-time trade simulation experience requires careful attention to the architecture, data handling, and computational efficiency. The GoQuant Trade Simulator applies multiple advanced performance optimization techniques to minimize latency, reduce resource consumption, and maintain a seamless user experience under continuous live data streams and frequent UI updates.

This section provides a detailed explanation of each optimization strategy, illustrated with relevant code snippets from the implementation.

Core Optimization Techniques

1. Memory Management
2. Network Communication
3. Data Structure Selection
4. Thread Management
5. Regression Model Efficiency

Detailed Implementation and Architectural Strategies

6. Asynchronous WebSocket Management via Dedicated Daemon Thread
7. Persistent Connection Management via Streamlit Session State
8. Single-Point Data Fetching and Reuse
9. Minimal Synthetic Dataset for Slippage Regression
10. Vectorized Numerical Computations with NumPy
11. Controlled UI Refresh and Lazy Rendering
12. Resilient External API Integration with Fallback Mechanism

1. Memory Management

Implementation:

- Use **Streamlit's session state** to persist objects like the WebSocket manager and async runner across reruns.
- Generate only a **small synthetic dataset** per iteration for regression instead of storing large historical data.

Justification:

- This prevents repeated initialization and reduces memory usage.
- Limits memory growth by avoiding accumulation of data points.

Code Snippet:

```
if "ws_manager" not in st.session_state:
    st.session_state.ws_manager = WebsocketManager(asset="BTC-USDT")
    st.session_state.runner = AsyncRunner(st.session_state.ws_manager)
    st.session_state.runner.start()
```

2. Network Communication

Implementation:

- The **WebSocket client runs asynchronously** on a separate thread to avoid blocking.
- Dynamic fee fetching uses **REST API calls with timeout and error handling**.
- Falls back to static fee tiers if the dynamic fetch fails.

Justification:

- Async thread allows UI to stay responsive.
- Proper error handling prevents app crashes or freezes.
- Reduces redundant network requests by caching fees per asset.

Code Snippet:

```
def fetch_okx_fee_rates(inst_id: str):
    try:
        response = requests.get("https://www.okx.com/api/v5/account/trade-fee", params={"instType"
        data = response.json()
        if data["code"] == "0" and data["data"]:
            fee_info = data["data"][0]
            maker_fee = -float(fee_info["maker"])
            taker_fee = -float(fee_info["taker"])
            return maker_fee, taker_fee
    except Exception:
        return None, None
```

3. Data Structure Selection

Implementation:

- Uses **NumPy arrays** for numerical time series and regression inputs.
- Uses **Pandas DataFrame** for UI-friendly tabular data display.
- Native Python lists/dictionaries for lightweight data organization.

Justification:

- NumPy enables efficient vectorized operations.
- Pandas integrates seamlessly with Streamlit's dataframe UI component.
- Minimal overhead structures chosen per use case.

Code Snippet:

```
timestamps = np.array([(now - datetime.timedelta(seconds=9 - i)).timestamp() for i in range(10)])
slippage_samples = np.random.uniform(low=0.1, high=1.0, size=10).reshape(-1, 1)
lr = LinearRegression()
lr.fit(timestamps, slippage_samples)
```

4. Thread Management

Implementation:

- WebSocket client runs inside a **daemon thread** controlled by an AsyncRunner class.
- Thread lifecycle management avoids duplicate threads and supports clean stopping.
- Thread runs the async event loop with asyncio.run.

Justification:

- Separates network I/O from UI thread, ensuring responsiveness.
- Prevents resource leaks and race conditions.
- Daemon threads allow the app to exit without hanging.

Code Snippet:

```
class AsyncRunner:
    def __init__(self, client: WebsocketManager):
        self.client = client
        self.thread = None

    def start(self):
        if self.thread is None or not self.thread.is_alive():
            self.thread = threading.Thread(target=self._run_loop, daemon=True)
            self.thread.start()

    def _run_loop(self):
        asyncio.run(self.client.connect())

    def stop(self):
        self.client.stop()
        if self.thread and self.thread.is_alive():
            self.thread.join(timeout=1)
```

5. Regression Model Efficiency

Implementation:

- Trains a **simple Linear Regression** model on only 10 synthetic data points every second.
- Uses timestamps as the only feature and slippage as the target.
- No hyperparameter tuning or complex model fitting, just a fast baseline estimate.

Justification:

- Minimal training data reduces computation and latency.
- Linear regression is lightweight and suitable for near real-time predictions.
- Balances model complexity and responsiveness in a live UI environment.

Code Snippet:

```
lr = LinearRegression()
lr.fit(timestamps, slippage_samples)
predicted_slippage = lr.predict(np.array([[now.timestamp()]]))[0][0]
```

6.Asynchronous WebSocket Management via Dedicated Daemon Thread

Explanation

Market data streaming via WebSocket is an inherently asynchronous, I/O-bound operation. Directly running this client in the main Streamlit thread would block UI rendering and cause the interface to freeze or lag. To circumvent this, the WebSocket client (WebsocketManager) is run inside a dedicated background thread using Python's threading and asyncio modules.

The AsyncRunner class encapsulates this logic. It creates a daemon thread that runs the asynchronous event loop executing the WebSocket connection and message handling logic.

How it Works

- The daemon thread runs independently of the Streamlit main thread.
- Uses asyncio.run() to manage the asynchronous WebSocket client within the thread.
- Allows Streamlit's event loop to remain free to update UI elements, handle user input, and refresh components.

Code Snippet

```
class AsyncRunner:
    def __init__(self, client: WebsocketManager):
        self.client = client
        self.thread = None

    def start(self):
        if self.thread is None or not self.thread.is_alive():
            self.thread = threading.Thread(target=self._run_loop, daemon=True)
            self.thread.start()

    def _run_loop(self):
        asyncio.run(self.client.connect())
```

Benefits

- **Non-blocking UI:** UI responsiveness is preserved even under heavy real-time data inflow.
- **Fault Isolation:** WebSocket failures or delays don't stall UI logic.
- **Scalability:** Allows easy scaling to multiple assets or connections without UI degradation.

7.Persistent Connection Management via Streamlit Session State

Explanation

Streamlit reruns the entire script on each user interaction or timer refresh. Without persistence, the WebSocket client and its managing thread would be recreated on every rerun, causing connection churn and resource overhead.

To solve this, WebSocket manager and its async runner instances are stored inside st.session_state, which persists across reruns during a user session.

How it Works

- On first run, the WebSocket client and runner are instantiated and stored.
- Subsequent reruns reuse existing instances from session state.
- When asset selection changes, old connections are cleanly stopped and new ones instantiated.

Code Snippet

```
if "ws_manager" not in st.session_state:
    st.session_state.ws_manager = WebsocketManager(asset="BTC-USDT")
    st.session_state.runner = AsyncRunner(st.session_state.ws_manager)
    st.session_state.runner.start()
```

Benefits

- **Resource Efficiency:** Prevents repeated network connections, saving bandwidth and CPU cycles.
- **Consistent Data Stream:** Maintains uninterrupted data flow and state.
- **Improved UX:** Faster UI updates with stable backend connections.

8.Single-Point Data Fetching and Reuse

Explanation

Market data snapshots are accessed once per UI update cycle and reused across multiple metrics computations and visualizations instead of repeatedly querying the WebSocket manager.

How it Works

- Fetch the latest data snapshot at the start of the update cycle.
- Use this snapshot consistently in metrics, charts, and calculations.

Code Snippet

```
latest_snapshot = st.session_state.ws_manager.get_data()

bids = latest_snapshot.get("bids", [])
asks = latest_snapshot.get("asks", [])
```

Benefits

- **Lower CPU Overhead:** Avoids redundant processing.
- **Data Consistency:** Guarantees all calculations within one update use the same market state.
- **Simpler Debugging:** Easier to trace and validate calculations with a single data source.

9.Minimal Synthetic Dataset for Slippage Regression

Explanation

To estimate slippage trends, linear regression is retrained every second on a very small synthetic dataset (10 samples). This approach balances computational speed and demonstration needs without requiring complex real historical data management.

How it Works

- Generates 10 synthetic slippage values uniformly at random within a range.
- Uses timestamped data points over the past 10 seconds as features.
- Fits a linear regression model from sklearn on this minimal dataset.

Code Snippet

```
timestamps = np.array([(now - datetime.timedelta(seconds=9 - i)).timestamp() for i in range(10)])
slippage_samples = np.random.uniform(low=0.1, high=1.0, size=10).reshape(-1, 1)

lr = LinearRegression()
lr.fit(timestamps, slippage_samples)
```

Benefits

- **Extremely Fast Training:** Model fitting completes in milliseconds.
- **Frequent Updates:** Enables near real-time slippage trend updates.
- **Low Memory Usage:** Small data footprint ideal for live, lightweight simulations.

10.Vectorized Numerical Computations with NumPy

Explanation

Mathematical formulas related to the Almgren-Chriss optimal execution model and other calculations are implemented using NumPy vectorized operations, which execute fast Cbacked computations rather than slow Python loops. **How it Works**

- Uses NumPy's sinh, sqrt, and array operations to compute optimal trade quantities over time intervals efficiently.
- Applies list comprehension only for concise array creation, leveraging NumPy for heavy math.

Code Snippet

```
sinh_sum = np.sinh(np.sqrt(risk_aversion * gamma / sigma**2) * T)
optimal_trades = [
    total_quantity * np.sinh(np.sqrt(risk_aversion * gamma / sigma**2) * (T - k * delta_t)) / sinh
    for k in range(num_intervals)
]
```

Benefits

- **Faster Computation:** Leverages compiled libraries under the hood.
- **Reduced CPU Load:** More efficient execution for potentially large parameter sets.
- **Scalability:** Handles large interval counts without slowing UI.

11.Controlled UI Refresh and Lazy Rendering

Explanation

Continuous UI rerendering can degrade performance if triggered too frequently or unnecessarily. This app uses two main approaches to balance freshness and efficiency:

1. **Throttled Refresh:** UI automatically refreshes every second with st_autorefresh to limit update frequency.
2. **Lazy Loading in Expanders:** Heavy visualizations and charts are enclosed in st.expander widgets, which only render content when expanded by the user. **How it Works**

- st_autorefresh triggers a script rerun every 1000 milliseconds.
- Visualization code blocks execute conditionally based on user interaction with expanders.

Code Snippet

```
from streamlit_autorefresh import st_autorefresh
st_autorefresh(interval=1000, key="auto_refresh")

with st.expander("📈 Visualize Market Data", expanded=True):
    # Visualization code here only runs when expanded
```

Benefits

- **Optimized Resource Use:** Limits CPU/GPU cycles for graph rendering.
- **Better UX:** Avoids UI lag and stuttering.
- **User Control:** Users decide when to load heavy content.

12.Resilient External API Integration with Fallback Mechanism

Explanation

Fetching live fee tiers from OKX's REST API includes handling unpredictable network delays and failures without impacting UI responsiveness.

How it Works

- API requests include short timeouts.
- Wrapped in try-except blocks to catch exceptions.

- On failure, predefined static fee tiers are used instead of dynamic data.

Code Snippet

```python
def fetch_okx_fee_rates(inst_id: str):
    try:
        response = requests.get(url, params=params, timeout=3)
        # Process response
    except Exception:
        return None, None
```

Benefits

- **Improved Stability:** Prevents UI hangs or crashes from external failures.
- **Graceful Degradation:** Provides best-effort dynamic data with fallback to static values.
- **Robust User Experience:** Reliable metrics regardless of external API health.

Summary

The GoQuant Trade Simulator combines:

- **Multithreading with async I/O** for continuous live data ingestion without UI blocking.
- **Session state persistence** to maintain stable connections across UI refreshes.
- **Efficient data reuse** to reduce overhead in calculations and rendering.
- **Minimal synthetic datasets** for rapid regression modeling.
- **Vectorized numeric computations** to accelerate mathematical modeling.
- **Smart UI update throttling and lazy visualization loading** for smooth interactivity.
- **Robust external API calls with fallbacks** to guarantee consistent user experience.

Together, these performance optimization approaches enable a highly responsive, scalable, and fault-tolerant trading simulation platform suitable for real-time financial analytics.