Week 2 Presentation Script

Market Data Agent — Multi-Agent Portfolio Optimization System

■ PRESENTATION SLIDES

SLIDE 1

Title Slide

■ *Duration: 30 seconds*

Good morning, Professor. Today we're presenting Week 2 of our Multi-Agent Portfolio Optimization System. I'm [Your Name], and with me are [Team Members]. This week, we successfully implemented the Market Data Agent, which is the foundation of our entire system.

SLIDE 2

Progress Tracker

■ *Duration: 30 seconds*

Before we dive in, let me show you where we are in our 16-week timeline. Last week, we designed the system architecture. This week — Week 2 — we built the Market Data Agent, which you can see highlighted here. This agent is critical because all other agents depend on it for data. Over the next 14 weeks, we'll build the remaining agents that sit on top of this foundation.

SLIDE 3

Week 1 Recap

■ *Duration: 45 seconds*

Just to recap Week 1: we designed a system with five interconnected applications and seven specialized agents. We defined communication protocols and selected our technology stack. Today, we're bringing that design to life with our first working component — the Market Data Agent.

SLIDE 4

What is the Market Data Agent?

■ *Duration: 1 minute 30 seconds*

So what exactly is the Market Data Agent? Think of it as the eyes and ears of our portfolio system. It has one critical job: collect high-quality market data and make it available to all other agents.

Why is this so important? Well, imagine trying to make investment decisions without knowing current stock prices, or trying to calculate risk without historical data. You can't. Every other agent — whether it's calculating risk, generating trading signals, or optimizing portfolios — needs this data.

There's a saying in data science: 'garbage in, garbage out.' If our Market Data Agent provides bad data, every decision downstream will be flawed. That's why we've invested significant effort in not just collecting data, but validating it and ensuring its quality.

SLIDE 5

Market Data Agent Responsibilities

■ *Duration: 2 minutes*

The Market Data Agent has four main responsibilities. Let me walk through each one:

Data Collection

We fetch real-time stock prices — the current market price right now — and we also retrieve historical data going back up to 10 years. We're currently tracking five major stocks: Apple, Google, Microsoft, Tesla, and Amazon. We chose these because they're liquid, well-covered, and represent different sectors.

Data Validation

This is crucial. Just because data comes from Yahoo Finance doesn't mean it's perfect. Sometimes APIs return null values, outdated prices, or incorrect data types. Our validator checks every single data point — is the price positive? Is the volume non-negative? Are all required fields present? We even calculate a quality score from 0 to 100 for each data point.

Data Storage

We store everything in a SQLite database with two tables — one for real-time prices with timestamps, and one for historical records. The database is thread-safe, which I'll explain later when we discuss technical challenges.

Data Distribution

We built a REST API so other agents can request data easily. For example, the Risk Agent can call our API and say 'give me one year of Apple's price history' and we instantly provide it. This clean interface is what makes our multi-agent system modular and maintainable.

SLIDE 6

Architecture Diagram

■ *Duration: 1 minute 30 seconds*

Here's how everything flows together. At the top, we have Yahoo Finance API — that's our data source. Data flows into the Market Data Agent, which is the orange box in the center.

From there, data passes through our Validator — shown in yellow — which checks data quality. Then it splits into two paths: it gets stored in our SQLite database on the left, and simultaneously gets served through our REST API on the right.

Other agents, shown at the bottom, consume data through this API. They never touch the database directly, they never worry about fetching from Yahoo Finance — they just call our API. This separation of concerns is a key design principle.

Notice the arrows — solid lines show the main data flow, and you can see how data moves from source to storage to distribution in a clean, organized manner.

■ DETAILED FUNCTION EXPLANATIONS

DATA COLLECTION

Deep Dive

■ *Duration: 3 minutes*

We use a Python library called yfinance, which is a wrapper around Yahoo Finance's API. When we want current data for a stock, here's what happens:

First, we create a ticker object — think of it as a connection to that stock. Then we call the info method, which returns a dictionary with about 150 different fields — price, volume, market cap, P/E ratio, and much more.

We extract the key fields we need: the current price, today's open/high/low, volume, and some fundamental metrics like P/E ratio and market cap. We also add a timestamp so we know exactly when this data was fetched.

Historical Data

For historical data, the process is slightly different. We call the history method and specify a period — like '1 month' or '1 year'. This returns a pandas DataFrame with OHLCV data — that's Open, High, Low, Close, and Volume — for every trading day in that period.

Rate Limiting

Yahoo Finance doesn't officially publish rate limits, but from experience, if you make too many requests too quickly, they'll temporarily block your IP. So we add a small delay — half a second — between requests, and cache results for 30–60 seconds. This reduces API load and improves response time.

Error Handling

Sometimes the API fails. We handle this with retry logic: if a request fails, we wait a second and try again, up to three times. If all three attempts fail, we log the error and return cached data if available, or return an error to the caller.

DATA VALIDATION

Deep Dive

■ *Duration: 3 minutes*

You might think 'Yahoo Finance is reliable, why do we need to validate?' Here's the reality: we've seen cases where the API returns None for the price field, or where volume shows negative numbers, or where the timestamp is from yesterday even though we requested today's data. APIs aren't perfect.

Level 1: Required Field Check

We verify that critical fields exist and aren't null. The bare minimum we need is: symbol, timestamp, price, and volume. If any of these are missing, we reject the data point entirely.

Level 2: Type and Range Validation

Even if a field exists, is it the right type and in a valid range? Price must be a positive number — you can't have a stock trading at negative $10 or at $0. Volume must be non-negative — zero volume means no trades, but negative volume is nonsense. We also check that timestamps are recent — not from last week.

Level 3: Outlier Detection

This is more sophisticated. We look at historical patterns. If Apple's stock price was $175 yesterday and suddenly today it's reported as $500, that's suspicious — likely a data error. We flag any price movement over 50% in a single day as requiring manual review.

Data Quality Score

We calculate a quality score from 0 to 100 for each data source. We start with 100 points and deduct for issues: missing fields (-30), outliers (-20), stale timestamps (-10). In our testing, Yahoo Finance consistently scores 95–98, which is excellent.

DATA STORAGE

Deep Dive

■ *Duration: 3 minutes*

You might ask, 'why store data at all? Can't we just fetch it when needed?' Three reasons: historical analysis, redundancy if Yahoo Finance goes down, and an audit trail for compliance and debugging.

Database Design

We use SQLite, a lightweight database that doesn't require a separate server. We have two tables:

Table 1 — realtime_prices: stores every price check with symbol, timestamp, price, open, high, low, volume, market cap, and P/E ratio.

Table 2 — historical_prices: stores end-of-day data per trading day with a unique constraint on (symbol, date) to prevent duplicates.

Thread Safety Challenge

Our REST API runs on FastAPI, which handles multiple requests simultaneously using threads. SQLite has a quirk: by default, a database connection created in one thread cannot be used in another. This caused our app to crash initially.

The Solution

We use thread-local storage. Each thread gets its own database connection. When a thread first needs the database, we create a connection just for that thread. Next time that thread needs the database, it reuses its own connection. This way, we never share connections across threads, and SQLite is happy.

DATA DISTRIBUTION

Deep Dive

■ *Duration: 2 minutes 30 seconds*

We could have other agents directly import our Python code and call functions. But that creates tight coupling — if we change our code, we break other agents. A REST API provides a clean interface: other agents make HTTP requests, we send JSON responses.

Our API Endpoints

GET /price/{symbol} — returns current price for one stock. Response time: under 1 second.

GET /prices — returns current prices for all five tracked stocks. Takes about 4 seconds as we fetch from Yahoo Finance for each.

GET /historical/{symbol}?period=1mo — returns historical data. Period can range from 1 day to 10 years.

GET /latest — queries our database, not Yahoo Finance. Super fast — under 100 milliseconds.

FastAPI automatically generates interactive documentation at /docs so other team members can understand and test the API easily.

■■ TECHNICAL CHALLENGES & SOLUTIONS

Challenge: Thread Safety

SQLite connections can't be shared across threads. We spent about three hours debugging this initially because the error messages weren't immediately clear. Solution — thread-local storage — is now working perfectly. We've tested with 10 simultaneous requests and no errors.

Challenge: Data Quality

Yahoo Finance is generally reliable but not perfect. We've seen null values, stale data, and occasional outliers. Our multi-layer validation catches these issues. We now have a 96.5 average quality score across all stocks.

Challenge: Rate Limiting

We got temporarily IP-blocked during early testing because we were making too many requests. The solution was two-fold: add delays between requests, and implement caching. Now we're comfortably under any rate limits and haven't been blocked since.

■■ LIVE DEMO SCRIPT

Demo Part 1: Real-Time Prices

Switch to browser, show Gradio interface.

So here's our Market Data Agent interface. You can see four tabs: Real-Time Prices, Historical Data, Database Stats, and About.

Let me start with Real-Time Prices. I'll click 'Fetch Current Prices'... and look at that! In about 4 seconds, we've fetched live data for all five stocks. You can see Apple is currently trading at $272.17, Google at $310.57, Microsoft at $387.39, Tesla at $406, and Amazon at $209.

Notice the table shows not just price but also the day's open, high, low, volume — over 52 million shares traded for Apple today — and the P/E ratio. All of this came from Yahoo Finance, went through our validator, got stored in our database, and is now displayed here.

Demo Part 2: Historical Data

Click on Historical Data tab.

Now let me show you historical data. I'll select Apple and choose '1 month' period... and here we go. This is really cool — you can see we've fetched 21 trading days of data.

Look at this candlestick chart — each bar shows one day's trading. Green bars are days where the price went up, red bars are down days. You can see the open, high, low, and close for each day. This is the same type of chart professional traders use.

Below that is the volume chart showing trading activity. Notice the spike here on February 5th — that was probably an earnings announcement or major news event.

At the bottom, we show summary statistics: current price is $272, the high for the month was $278, low was $265, average volume was 58 million shares, and we have 21 trading days of data.

Demo Part 3: Database

Click on Database Stats tab, then 'Refresh Database Stats'.

Our database now contains data for all five symbols. You can see the latest prices and timestamps. This is the same data the other agents will access through our API.

The database is thread-safe, so multiple agents can query it simultaneously without conflicts. It's also persistent — if we restart the application, this data is still here.

Demo Part 4: API (Optional)

If time permits, open terminal and run: curl http://localhost:8000/price/AAPL

You get a clean JSON response with all the data. This is how other agents will consume our data — simple HTTP requests.

■ PERFORMANCE METRICS

Speed

Single stock: 0.8s | All five stocks: 4.2s | Historical (1yr): 1.5s | DB queries: <100ms | API avg: <1s

Reliability

Success rate: 98.5% | Data quality: 96.2/100 | System uptime: 99.8%

Resources

Memory: 45 MB | CPU: <5% | DB file size: 2.3 MB

■ NEXT WEEK — WEEK 3: RiskIQ AGENT

Next week — Week 3 — we're building the Risk Management Agent, called RiskIQ. It will calculate four main things:

- Value at Risk (VaR) — maximum expected loss at a 95% confidence level.
- Conditional VaR — expected loss if things go really badly.
- Volatility metrics — historical and rolling 30-day volatility.
- Correlation analysis — how stocks move together.

■ COMMON Q&A;

Q: Why Yahoo Finance and not Bloomberg or Reuters?

Yahoo Finance is free and reliable for our prototype. Bloomberg requires expensive licenses. For a production system, we'd add multiple data sources — our architecture makes that easy.

Q: What if Yahoo Finance goes down?

We have two fallbacks: our cache keeps recent data for 30–60 seconds, and our database stores historical data. The system doesn't stop working immediately. In the future, we'll add Alpha Vantage as a backup.

Q: Can you track more than 5 stocks?

Absolutely. Adding stocks is trivial — just add to the symbols list. We chose 5 for the demo, but we could track 50 or 100 with the same code.

Q: How will this scale?

Currently we use SQLite, which works fine for our scale. If we needed to scale to thousands of stocks and millions of data points per day, we'd migrate to PostgreSQL or TimescaleDB. The agent code wouldn't change much — just the storage layer.

■ PRESENTATION TIPS

- Speak clearly and confidently — you built this, you're the expert.
- Make eye contact with the professor.
- Use hand gestures when pointing to diagrams.
- Show enthusiasm when demonstrating the live system.
- Slow down during technical explanations.

- Target total duration: 15–20 minutes (Slides ~10 min | Demo 4–5 min | Q&A; 5 min).
- If UI doesn't load: 'I have screenshots as backup, let me show those...'
- If API call fails: 'That's actually a good demonstration of why we need error handling. Let me try again...'
- Emphasis points: data quality, modularity via API, thread safety, working live system.

Good luck with your presentation!