Stock Analysis

Project 4
11/10/2025

Name	**Email Address**
Aaron Downing	aaron.downing652@topper.wku.edu
Ryerson Brower	ryerson.brower178@topper.wku.edu

CS 396
Fall 2025
Project Technical Documentation

Contents

List of Figures

1 Introduction

1.1 Project Overview

This project implements a real-time stock analysis pipeline that fuses live market prices with company-level fundamental data and exposes the combined analysis through an interactive web dashboard. The overall goal is to simulate a lightweight, production-style analytics stack using containerized microservices and a shared database, while demonstrating concepts such as data in motion, data fusion, technical indicators, and infrastructure-as-code orchestration.

The way the system is organized into three Python based microservices and a separate frontend. The price polling service periodically retrieves recent, historical and intraday price data for a small universe tickers (AAPL, GOOGL, and MSFT) using the public yfinance API. This data is normalized into a consistent schema and stored in a MySQL database. The Fundamental Data Service uses the same API to fetch key company metrics such as market capitalization, price-to-earnings ratio, and dividend yield, and persists these in a separate fundamentals table.

The Analysis and Visualization Service connects to the same MySQL instance and performs data fusion by joining the time-series price data with the latest fundamental snapshot for a selected ticker. On top of the fused dataset, the service computes technical indicators including a 20-period Simple Moving Average (SMA) and a 14-period Relative Strength Index (RSI). These metrics are exposed via a REST API endpoint consumed by a React-based dashboard.

The frontend dashboard allows a user to enter or select a ticker and view, on a single page, the recent price history, overlayed technical indicators, and current fundamental metrics. All services run in separate Docker containers and are orchestrated through Docker Compose, which brings up the database, backend services, and web frontend as a single reproducible stack.

1.2 Project Scope

The scope of this project is to design, implement, and containerize a small but realistic stock analytics pipeline that operates on near real-time data and demonstrates end-to-end data flow from external APIs to a user-facing dashboard. The system focuses on three equities—Apple (AAPL), Alphabet (GOOGL), and Microsoft (MSFT)—to keep the data volume manageable while still illustrating multi-ticker support.

From a deliverables perspective, the project includes: A Price Polling Service that continuously fetches and stores both daily and 5-minute intraday price data using yfinance, handles duplicate records via ON DUPLICATE KEY UPDATE, and exposes a /api/prices/¡ticker¿ endpoint for downstream consumers. A Fundamental Data Service that retrieves and persists company fundamentals (market cap, P/E ratio, dividend yield, and last-updated timestamp) and exposes them via /api/fundamentals/¡ticker¿. An Analysis and Visualization Service that: Reads from the `price_data` and `fundamentals` tables, Calculates SMA(20) and RSI(14), Fuses the datasets into a single JSON payload, And exposes that via /api/data/¡ticker¿ for the frontend. A React/Vite frontend that calls the analysis API, renders an interactive price chart with technical indicators, and displays the most recent fundamentals in a table.

The project also includes a Docker Compose configuration that defines and orchestrates the containers for all backend services, the MySQL database, and the frontend application. Non-functional aspects within scope include basic fault tolerance (retrying database connections on startup), logging of background polling loops, and performance tuned to keep data no more than roughly 5–15 minutes out of date.

Out of scope are advanced production concerns such as authentication, scaling across multiple nodes, comprehensive alerting/monitoring, and full-featured order execution or portfolio management. However, the modular architecture is intentionally designed so that additional indicators, more tickers, or different data sources could be added with minimal changes to the existing services.

1.3 Technical Requirements

1.3.1 Functional Requirements

The system shall be implemented using at least three distinct Dockerized microservices: a Price Polling Service, a Fundamental Data Service, and an Analysis and Visualization Service. The Price Polling Service shall

Mandatory Functional Requirements
The project must be implemented using at least three distinct Docker containers
The Price Polling Service
The Fundamental Data Service
The Analysis and Visualization Service
This service must perform Data Processing by calculating at least one technical indicator for each stock
The Analysis and Visualization Service must host a visual dashboard

automatically fetch time-series stock price data for at least three tickers (AAPL, GOOGL, MSFT), including daily and 5-minute intraday intervals, using a public financial API (`yfinance`). The Fundamental Data Service shall automatically fetch company-specific metrics (market capitalization, trailing P/E ratio, dividend yield, and last-updated timestamp) for the same three tickers, using `yfinance`. The Analysis and Visualization Service shall consume data from both the price and fundamental services (via the shared MySQL database), and perform data fusion by combining the latest fundamentals with recent historical price data for a selected ticker. The Analysis and Visualization Service shall compute at least one technical indicator per stock, including a 20-period Simple Moving Average (SMA) and a 14-period Relative Strength Index (RSI), based on the close prices. The frontend dashboard shall allow a user to select or enter a stock ticker and view its recent time-series price data, technical indicators (SMA, RSI), and latest fundamental metrics on one unified screen.

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
The entire 3-container application stack must be automated
The data schemas for all data in motion and the complete data pipeline must be clearly defined, and visualized
The system must include robust error handling for API failures
The price data on the dashboard should be "near real-time," with data being no more than 15 minutes old.
The dashboard must be responsive, clear, and must load in under 10 seconds.
All communication between containers must be clearly defined

All core services (price polling, fundamentals, analysis/visualization, frontend, and database) shall run as separate Docker containers, orchestrated using Docker Compose as the infrastructure-as-code tool. The schemas for all data in motion (price data, fundamental data, fused data) and the end-to-end data pipeline shall be clearly defined and documented, including table definitions and JSON payload structures used between services. Each service shall implement basic error handling for database connectivity issues and API failures (e.g., network problems, unavailable data), logging errors to standard output without causing the service to crash. The system shall maintain near real-time price data, with the latest intraday prices being no more than approximately 15 minutes old (via a 5-minute polling schedule and 5-minute intraday interval). The web dashboard shall render the selected stock's data and charts in under 10 seconds under typical conditions, assuming the backend services and database are running and pre-populated. Communication patterns between containers shall be clearly defined: backend services and the dashboard communicate via HTTP REST APIs, while the price and fundamental services communicate their outputs to the analysis service through a shared MySQL database volume.

2 Project Modeling, Design, and System Architecture

2.1 Data Sources

This system uses the following data sources: Public Financial API via `yfinance` (Price Polling Service). The price-polling microservice uses yfinance.Ticker(ticker).history(...) to obtain: Daily OHLCV data for the past 60 days (period="60d", interval="1d"), Intraday OHLCV data at 5-minute intervals for the most recent trading day (period="1d", interval="5m"). Public Financial API via yfinance (Fundamental Data Service) The fundamental-data microservice uses yfinance.Ticker(ticker).info to retrieve: marketCap (market capitalization), trailingPE (P/E ratio), dividendYield, And other metadata, from which only relevant fields are extracted.

2.2 Data Formats and Schemas

All services share a single MySQL database named `stocks`. The two core tables are: The Price data table `ticker` (VARCHAR): Stock symbol, e.g., "AAPL". `Timestamp` (DATETIME): Timestamp of the price bar (daily or 5-minute). `open_price` (DOUBLE): Opening price. `high_price` (DOUBLE): High price for the bar. `Low_price` (DOUBLE): Low price for the bar. `close_price` (DOUBLE): Closing price. volume— (DOUBLE): Trading volume. The service uses an `ON DUPLICATE KEY UPDATE` clause with a unique key on `(ticker, timestamp)` to prevent duplicate records while updating values when new data arrives.Then there is the Fundamentals table `ticker` (VARCHAR): Stock symbol. `Market_cap` (DOUBLE or BIGINT): Market capitalization. `pe_ratio` (DOUBLE): Trailing P/E ratio. `dividend_yield` (DOUBLE or NULL): Dividend yield. `last_updated` (DATETIME): Timestamp when the fundamental metrics were last fetched. A unique constraint on `ticker` (or `(ticker, last_updated)`) is used in conjunction with `ON DUPLICATE KEY UPDATE` to keep the most recent snapshot.

API JSON Schemas Price service: `GET /api/prices/<ticker>` → JSON array of objects with fields mirroring the `price_data` columns. Fundamentals service: `GET /api/fundamentals/<ticker>` → JSON array of objects with the columns from `fundamentals`. Analysis/visualization service: `GET /api/data/<ticker>` → JSON object of the form:

2.3 Data Storage

All persistent storage is handled by a single MySQL instance running in its own Docker container. The database uses a named Docker volume (e.g., mysqldata) to persist data across container restarts. The Price Polling and Fundamental Data services write into the stocks database, while the Analysis and Visualization service is read-only against this database. There are no additional file-based storage systems; all durable state flows through MySQL. The React frontend does not store data permanently; it only holds the most recently fetched payloads in browser memory (React state) for visualization.

3 Data Pipeline and Movement

3.1 Pipeline Visualization

The pipeline follows a linear yet cyclical flow, where new data continuously propagates forward through the architecture:

1. **External Financial API (Yahoo Finance via yfinance)** ↓ (HTTPS request/response)
2. **Price Polling Service (Docker Container)** ↓ (SQL insert/update via TCP to MySQL)
3. **Fundamental Data Service (Docker Container)** ↓ (SQL insert/update via TCP to MySQL)
4. **MySQL Database (Persistent Shared Storage Volume)** ↓ (SQL select via TCP to MySQL)
5. **Analysis & Visualization Service (Docker Container)** ↓ (JSON API response via HTTP)
6. **React Web Dashboard (Browser Client)**

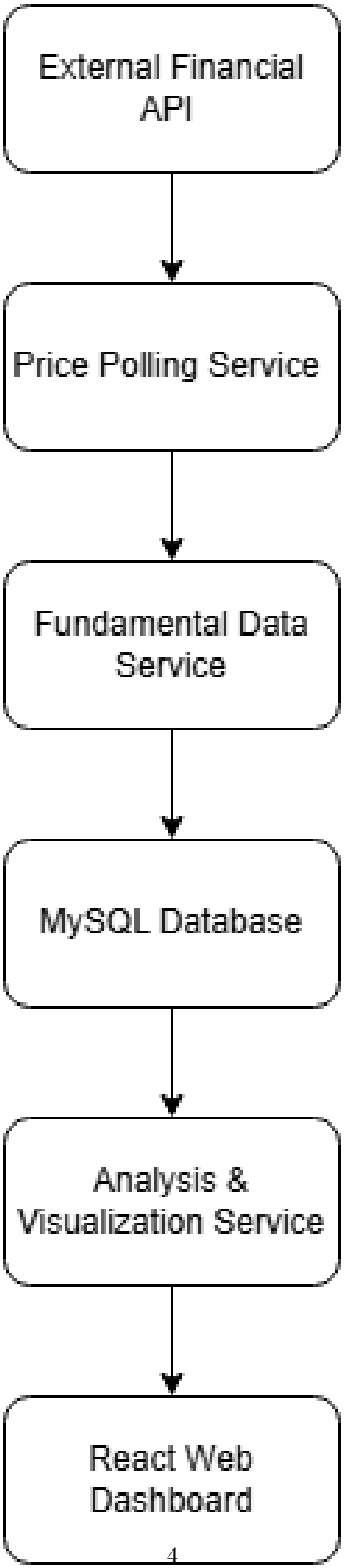
The system is event-loop driven rather than user-triggered: the price service polls every 5 minutes and the fundamental service updates once per day. The dashboard retrieves fused data only when requested by the user.

3.2 Data Movement

4 Data Processing and Fusion

4.1 Data Fusion

Data fusion in this project refers to the linking of two heterogeneous financial datasets: Dynamic data (time-series price fluctuations) and Static data (company fundamentals). Fusion occurs within the Analysis and Visualization Service. After retrieving the latest fundamental record from the `fundamentals` table, the service appends


External Financial API
Price Polling Service
Fundamental Data Service
MySQL Database
Analysis & Visualization Service
React Web Dashboard

Data Transfer Step	Producer	Consumer	Transport Method	Protocol	Trigger
Fetch raw market price data	Yahoo Finance	Price Service	API Request	HTTPS	Every 5 min
Insert processed price data	Price Service	MySQL	SQL Write	TCP	Each polling cycle
Fetch raw fundamentals	Yahoo Finance	Fundamentals Service	API Request	HTTPS	Every 24 hours
Insert fundamentals	Fundamentals Service	MySQL	SQL Write	TCP	Each fetch cycle
Retrieve fused dataset	Analysis Service	MySQL	SQL Read	TCP	API request
Deliver fused analytics	Analysis Service	Dashboard	REST API	HTTP	User request

this information to every row of the price dataset returned to the dashboard. This guarantees that when a user views a stock, the charted time-series data is accompanied by the most recent business context (e.g., market cap, P/E ratio, dividend yield). Fusion ensures that a stock is not interpreted purely by price trends nor solely by fundamentals, but from a complete analytical perspective.

4.2 Data Processing and Transformation

Before any visualization, multiple transformations prepare the time-series data for analysis: Normalization: Raw API data is converted to a consistent schema: timestamp, open, high, low, close, volume. Timestamps are converted to Python `datetime` objects and stored in MySQL. Duplicate Conflict Resolution: Since both daily and intraday data may contain overlapping timestamps, the insert statements use ON DUPLICATE KEY UPDATE, ensuring the database always contains the most accurate entry. Technical Indicator Computation: Two primary analytics are calculated using closing prices: 20-period Simple Moving Average (SMA) 14-period Relative Strength Index (RSI). These are added as new columns in the price dataset passed to the dashboard. Filtering of Invalid Rows: Price rows where rolling indicators are not yet defined (e.g., the first 20 rows for SMA) are removed to prevent chart distortion. The result is a clean, enriched, and analysis-ready dataset delivered through the API.

5 Model and Visualization

5.1 Model Implementation

5.2 Visualization Approaches

Instead of a predictive AI model, this project implements a technical-indicator analytics model containerized inside the Analysis and Visualization Service. The model performs: Time-series fusion with fundamentals, SMA(20) calculation, RSI(14) calculation, and Export of the fused dataset via REST API\textbf{API Specification:} `GET /api/data/<ticker>` → returns a JSON payload: "ticker": "...","`price_data`": [timestamp, close, volume, SMA, RSI, ...], "fundamentals": `market_cap`, `pe_ratio`, `dividend_yield`, `last_updated`

6 Appendix

6.1 Software Product Build Instructions

This section outlines all steps required to set up the Stock Analysis System on a new development machine. These instructions assume basic familiarity with Docker, Python, and MySQL.

Prerequisites

- Docker and Docker Compose installed.
- Python 3.10+ installed (optional, only needed if developing locally outside Docker).
- Git installed.

Project Setup

1. Clone the repository:

```
git clone https://github.com/RyersonBrower/StockAnalysis
cd StockAnalysis
```

2. Ensure the project folder structure is intact:
   - `price-polling/`
   - `fundamental-data/`
   - `analysis-visualization/`
   - `frontend/` (if included)
   - `mysql_data/` (auto-created by Docker)
   - `docker-compose.yml`
3. Build and start all services:

```
docker compose up --build
```

4. Docker will launch the following containers:
   - **mysql**: Runs the MySQL database on port 3307:3306.
   - **price-polling**: Pulls real-time stock prices and stores them in MySQL.
   - **fundamental-data**: Fetches company fundamentals and stores them.
   - **analysis-visualization**: Performs data fusion and technical analysis.
   - **frontend** (optional): Web UI for charts and metrics.
5. Once all containers show "ready," the system is fully operational.

Database Initialization

1. On first startup, the MySQL container initializes the database using the credentials defined in `docker-compose.yml`:
   - Username: `appuser`
   - Password: `apppassword`
2. Tables are created automatically by the Python services if they do not already exist.

Updating the Software

1. Pull latest changes:

```
git pull
```

2. Rebuild containers:

```
docker compose up --build
```

Shutting Down the System

```
docker compose down
```

6.2 Software Product User Guide

This section provides an overview for both general users and administrative users of the Stock Analysis System.

General User Guide

- Access the system through the frontend dashboard (if enabled), typically at:

  ```
  http://localhost:3000
  ```

- View real-time stock prices, charts, and computed technical indicators such as:
  - RSI (Relative Strength Index)
  - Moving Averages (SMA/EMA)
  - Volume trends
  - Price history visualizations
- Use the search bar to lookup any supported ticker symbol.
- Charts automatically update as the backend polling service gathers new data.
- No installation or configuration is required beyond running the Docker stack.

Administrative User Guide

Administrative users have additional responsibilities and capabilities:

- **Monitor Container Health:**

  ```
  docker ps
  ```

- **Check logs for debugging:**

  ```
  docker logs price-polling
  docker logs fundamental-data
  docker logs analysis-visualization
  ```

- **Access the MySQL database:**

  ```
  mysql -h 127.0.0.1 -P 3307 -u appuser -p
  ```

- **Inspect stored data:**

  ```
  SELECT * FROM price_data LIMIT 20;
  SELECT * FROM fundamentals;
  ```

- **Restart individual services:**

  ```
  docker compose restart price-polling
  ```

- **Modify stock symbols:** Update your service configuration files inside `price-polling/` or `fundamental-data/` and rebuild containers.

6.3 Source Code with Comments

6.4 Analysis-Visualization Container

app.py

```
import os
import sys
import time
import pandas as pd
import mysql.connector
from mysql.connector import Error
from flask import Flask, jsonify
from flask_cors import CORS
import threading

app = Flask(__name__)
CORS(app)

# ------------- Database Config -------------------
host=os.environ.get("MYSQL_HOST", "mysql")
port=int(os.environ.get("MYSQL_PORT", 3306))
user=os.environ.get("MYSQL_USER", "appuser")
password=os.environ.get("MYSQL_PASSWORD", "apppassword")
database=os.environ.get("MYSQL_DATABASE", "stocks")

# ------------- Wait for MySQL to be ready ----------
for i in range(20):
    try:
        connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
        )
        print("Connected to MySQL")
        connection.close()
        break
    except Error:
        print(f"MySQL not ready, retrying ... ({i+1}/20)")
        time.sleep(2)
else:
    raise Exception("Cannot connect to MySQL after multiple retries")


# ---------------- Query Functions ----------------
def get_connection():
    return mysql.connector.connect(
        host=host,
        port=port,
        user=user,
        password=password,
        database=database,
    )
```

```
def get_price_data(ticker, limit=80):
    connection = get_connection()
    query = """
        SELECT timestamp, open_price, high_price, low_price, close_price, volume
        FROM price_data
        WHERE ticker = %s
        ORDER BY timestamp DESC
        LIMIT %s
    """
    df = pd.read_sql(query, connection, params=(ticker, limit))
    connection.close()
    return df[::-1]

def get_fundamentals(ticker):
    connection = get_connection()
    query = """
        SELECT pe_ratio, market_cap, dividend_yield, last_updated
        FROM fundamentals
        WHERE ticker = %s
        ORDER BY last_updated DESC
        LIMIT 1
    """
    cursor = connection.cursor(dictionary=True)
    cursor.execute(query, (ticker,))
    row = cursor.fetchone()
    cursor.close()
    connection.close()
    return row if row else {}


# ------------------- Indicator Calculation ----------------
def calculate_sma(df, window=20):
    df["SM20"] = df["close_price"].rolling(window=window).mean()
    return df

def calculate_rsi(df, window=14):
    delta = df["close_price"].diff()
    gain = delta.clip(lower=0)
    loss = -delta.clip(upper=0)
    avg_gain = gain.rolling(window=window, min_periods=window).mean()
    avg_loss = loss.rolling(window=window, min_periods=window).mean()
    rs = avg_gain / avg_loss
    df["RSI"] = 100 - (100 / (1 + rs))
    return df


# ----------------------- Data Fusion ------------------------
def fuse_data(ticker):
    price_df = get_price_data(ticker)
    fundamentals = get_fundamentals(ticker)

    if price_df.empty or not fundamentals:
```

```
        print(f"No data available for {ticker}")
        return None

    price_df = calculate_sma(price_df)
    price_df = calculate_rsi(price_df)
    price_df = price_df.dropna(subset=["SM20", "RSI"])

    return {
        "ticker": ticker,
        "price_data": price_df.to_dict(orient="records"),
        "fundamentals": fundamentals
    }


# ------------- Flask API Routes -----------------------
@app.route("/api/data/<ticker>")
def get_combined_data(ticker):
    fused = fuse_data(ticker)
    if fused is None:
        return jsonify({"error": "Data not available"}), 404
    return jsonify(fused)


# --------------- Background Monitoring (Optional) -------------
def periodic_health_check():
    sys.stdout.reconfigure(line_buffering=True)
    while True:
        print("Health check: verifying analysis data...", flush=True)
        for ticker in ["AAPL", "GOOGL", "MSFT"]:
            df = get_price_data(ticker)
            print(f"{ticker}: {len(df)} rows of price data.", flush=True)
        time.sleep(600)


# ---------------- Run Flask + Background Thread ---------------
if __name__ == "__main__":
    threading.Thread(target=periodic_health_check, daemon=True).start()
    print("Starting analysis & Visualization Flask server...", flush=True)
    app.run(host="0.0.0.0", port=5003)
```

Dockerfile

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5003
CMD ["python", "app.py"]
```

requirements.txt

```
flask
flask-cors
```

```
mysql-connector-python
matplotlib
pandas
```

6.5 Frontend (React Application)

src/App.css

```
.app-container {
  font-family: sans-serif;
  padding: 2rem;
  background-color: #fafafa;
  color: #222;
  min-height: 100vh;
  width: 95vw;
}

h1 {
  text-align: center;
  margin-bottom: 2rem;
}

.input-section {
  display: flex;
  justify-content: center;
  align-items: center;
  margin-bottom: 1rem;
}

.input-section label {
  margin-right: 0.5rem;
  font-weight: 500;
}

.input-section input {
  padding: 0.4rem 0.6rem;
  font-size: 1rem;
  border-radius: 6px;
  border: 1px solid #ccc;
  width: 120px;
  text-align: center;
}

.error {
  color: red;
  text-align: center;
  margin-top: 1rem;
}

.data-section {
  margin-top: 2rem;
}

.fundamentals table,
```

```
.price-data table {
  border-collapse: collapse;
  width: 100%;
}

.fundamentals td.key,
.price-data th,
.price-data td {
  border: 1px solid #ddd;
  padding: 0.5rem .7rem;
  text-align: left;
}

.fundamentals td.key {
  font-weight: bold;
  background-color: #f5f5f5;
}

.table-container {
  max-height: 400px;
  overflow-y: auto;
  border: 1px solid #ccc;
  background: white;
}

.price-data th {
  background-color: #f0f0f0;
  position: sticky;
  top: 0;
  z-index: 1;
}

.charts {
  margin-top: 2rem;
  display: grid;
  gap: 2rem;
}

.chart-container {
  background: #fff;
  border-radius: 12px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
  padding: 1rem;
}

.loading {
  color: #007bff;
  font-weight: bold;
  margin-bottom: 1rem;
}

.error {
  color: #d9534f;
```

```
  font-weight: bold;
  margin-bottom: 1rem;
}

.no-data {
  color: #555;
  font-style: italic;
  margin-top: 1rem;
}

.chart-section {
  margin: 2rem 0;
}

input {
  transition: border-color 0.2s ease;
}

input:focus {
  border-color: #007bff;
  outline: none;
}
```

src/App.jsx

```
import { useEffect, useState } from "react";
import {
  LineChart,
  Line,
  XAxis,
  YAxis,
  Tooltip,
  CartesianGrid,
  ResponsiveContainer,
} from "recharts";
import "./App.css";

function App() {
  const [data, setData] = useState(null);
  const [ticker, setTicker] = useState("AAPL");
  const [debouncedTicker, setDebouncedTicker] = useState("AAPL");
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  // Debounce input (1 second)
  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedTicker(ticker);
    }, 1000);
    return () => clearTimeout(handler);
  }, [ticker]);

  // Fetch data
```

```
useEffect(() => {
  if (!debouncedTicker) return;

  setLoading(true);
  setError(null);

  fetch(`${import.meta.env.VITE_API_URL}/api/data/${debouncedTicker}?limit=80`)
    .then((res) => {
      if (!res.ok) throw new Error("Failed to fetch data");
      return res.json();
    })
    .then(setData)
    .catch((err) => {
      console.error(err);
      setError("No data available or API error");
      setData(null);
    })
    .finally(() => setLoading(false));
}, [debouncedTicker]);

return (
  <div className="app-container">
    <h1> Stock Analysis Dashboard</h1>

    <div className="input-section">
      <label>Enter Ticker: </label>
      <input
        value={ticker}
        onChange={(e) => setTicker(e.target.value.toUpperCase())}
      />
    </div>

    {loading && <p className="loading">Loading...</p>}
    {!ticker && <p className="error">{error}</p>}

    {data ? (
      <div className="data-section">

        {/* Fundamentals */}
        <div className="fundamentals">
          <h2>Fundamentals</h2>
          <table>
            <tbody>
              {Object.entries(data.fundamentals).map(([key, value]) => (
                <tr key={key}>
                  <td className="key">{key}</td>
                  <td className="value">{value}</td>
                </tr>
              ))}
            </tbody>
          </table>
        </div>
```

```
{/* Price Chart */}
<div className="chart-section">
  <h2>Price Trend (Close Price)</h2>
  <ResponsiveContainer width="100%" height={300}>
    <LineChart
      data={data.price_data.slice(-20)}
      margin={{top: 20, right: 30, left: 0, bottom: 5}}
    >
      <CartesianGrid strokeDasharray="3 3" />
      <XAxis dataKey="timestamp" tick={false} />
      <YAxis domain={["auto", "auto"]}/>
      <Tooltip />
      <Line
        type="monotone"
        dataKey="close_price"
        stroke="#007bff"
        dot={false}
        name="Close"
      />
      <Line
        type="monotone"
        dataKey="SM20"
        stroke="#ff7300ff"
        dot={false}
        name="SMA20"
      />
    </LineChart>
  </ResponsiveContainer>
</div>

{/* Price Table */}
<div className="price-data">
  <h2>Price Data (last 20 rows)</h2>
  {data.price_data.length === 0 ? (
    <p>No price data available</p>
  ) : (
    <div className="table-container">
      <table>
        <thead>
          <tr>
            <th>Timestamp</th>
            <th>Open</th>
            <th>High</th>
            <th>Low</th>
            <th>Close</th>
            <th>Volume</th>
            <th>SM20</th>
          </tr>
        </thead>
        <tbody>
          {data.price_data.slice(-20).map((row, idx) => (
            <tr key={idx}>
              <td>{row.timestamp}</td>
```

```
                        <td>{row.open_price}</td>
                        <td>{row.high_price}</td>
                        <td>{row.low_price}</td>
                        <td>{row.close_price}</td>
                        <td>{row.volume}</td>
                        <td>{row.SM20 ?? "|"}</td>
                      </tr>
                    ))}
                  </tbody>
                </table>
              </div>
            )}
          </div>

        </div>
      ) : (
        !loading &&
        !error && <p className="no-data">Enter a ticker to view stock data.</p>
      )}
    </div>
  );
}

export default App;
```

src/main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

6.5.1 fundamental-data Service

app.py

```
import os
import sys
import time
import pandas as pd
import yfinance as yf
import mysql.connector
from mysql.connector import Error
from datetime import datetime
from flask import Flask, jsonify
import threading
```

```
app = Flask(__name__)

# --------------- Database Config -------------------
host=os.environ.get("MYSQL_HOST", "localhost")
port=int(os.environ.get("MYSQL_PORT", 3306))
user=os.environ.get("MYSQL_USER", "appuser")
password=os.environ.get("MYSQL_PASSWORD", "apppassword")
database=os.environ.get("MYSQL_DATABASE", "stocks")

# -------------Wait for MySQL to be ready --------------
for i in range(20):  # try 20 times
    try:
        connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
        )
        print("Connected to MySQL")
        break
    except Error as e:
        print(f"MySQL not ready, retrying... ({i+1}/20)")
        time.sleep(2)
else:
    raise Exception("Cannot connect to MySQL after multiple retries")


# --------------------- Helper Functions --------------------

def get_connection():
    return mysql.connector.connect(
        host=host,
        port=port,
        user=user,
        password=password,
        database=database,
    )


def insert_fundamentals(ticker, market_cap, pe_ratio, dividend_yield, last_updated):

    connection = get_connection()
    cursor = connection.cursor()

    sql = """
        INSERT INTO fundamentals(ticker, market_cap, pe_ratio, dividend_yield, last_updated)
        VALUES(%s, %s, %s, %s, %s)
        ON DUPLICATE KEY UPDATE
            ticker = VALUES(ticker),
            market_cap = VALUES(market_cap),
            pe_ratio = VALUES(pe_ratio),
            dividend_yield = VALUES(dividend_yield),
            last_updated = VALUES(last_updated)
```

```
    """

    values = (ticker, market_cap, pe_ratio, dividend_yield, last_updated)

    cursor.execute(sql, values)
    connection.commit()
    cursor.close()
    connection.close()

    print(f"\n Last Updated at {last_updated}")


def fetch_and_store():
    tickers = ["AAPL", "GOOGL", "MSFT"]

    for ticker in tickers:
        stock = yf.Ticker(ticker)
        info = stock.info

        market_cap = info.get("marketCap")
        pe_ratio = info.get("trailingPE")
        dividend_yield = info.get("dividendYield")
        last_updated = datetime.now()

        print(f"\n--- {ticker} ---")
        print(f"Market Cap: {market_cap}, PE Ratio: {pe_ratio}, Divident Yield: {dividend_yield}")

        insert_fundamentals(ticker, market_cap, pe_ratio, dividend_yield, last_updated)
```

Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 5002
CMD ["python", "app.py"]
```

requirements.txt

```
flask
mysql-connector-python
requests
pandas
yfinance
```

6.5.2 price-polling Service

app.py

```
import os
import sys
import time
import yfinance as yf
import pandas as pd
```

```
import mysql.connector
from mysql.connector import Error
from flask import Flask, jsonify, request
import threading
import datetime

app = Flask(__name__)

# --------------- Database Config -------------------
host=os.environ.get("MYSQL_HOST", "localhost")
port=int(os.environ.get("MYSQL_PORT", 3306))
user=os.environ.get("MYSQL_USER", "appuser")
password=os.environ.get("MYSQL_PASSWORD", "apppassword")
database=os.environ.get("MYSQL_DATABASE", "stocks")

# -------------Wait for MySQL to be ready --------------
for i in range(20):  # try 20 times
    try:
        connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
        )
        print("Connected to MySQL")
        break
    except Error as e:
        print(f"MySQL not ready, retrying... ({i+1}/20)")
        time.sleep(2)
else:
    raise Exception("Cannot connect to MySQL after multiple retries")

#---------------Helper Functions-----------------

def get_connection():
    return mysql.connector.connect(
        host=host,
        port=port,
        user=user,
        password=password,
        database=database,
    )

def insert_price_data(ticker, data):
    connection = get_connection()
    cursor = connection.cursor()

    sql = """
        INSERT INTO price_data (ticker, timestamp, open_price, high_price, low_price, close_price, vo
        VALUES (%s, %s, %s, %s, %s, %s, %s)
        ON DUPLICATE KEY UPDATE
            open_price = VALUES(open_price),
            high_price = VALUES(high_price),
```

```
            low_price = VALUES(low_price),
            close_price = VALUES(close_price),
            volume = VALUES(volume)
    """

    for idx, row in data.iterrows():
        timestamp = idx.to_pydatetime()
        values = (
            ticker,
            timestamp,
            float(row["Open"]),
            float(row["High"]),
            float(row["Low"]),
            float(row["Close"]),
            float(row["Volume"]),
        )
        cursor.execute(sql, values)

    connection.commit()
    cursor.close()
    connection.close()

    latest_row = data.tail(1)
    print(f"\n--- {ticker} Latest Row ---")
    print(latest_row)
    print(f"{ticker} data inserted.\n")

def fetch_and_store():
    tickers = ["AAPL", "GOOGL", "MSFT"]

    for ticker in tickers:
        stock = yf.Ticker(ticker)

        historical = stock.history(period="60d", interval="1d")
        if not historical.empty:
            historical = historical[["Open", "High", "Low", "Close", "Volume"]]
            insert_price_data(ticker, historical)
            print(f"{ticker} historical daily data inserted.")
        else:
            print(f"No historical data for {ticker}")

        intraday = stock.history(period="1d", interval="5m")
        if not intraday.empty:
            intraday = intraday[["Open", "High", "Low", "Close", "Volume"]]
            insert_price_data(ticker, intraday)
            print(f"{ticker} intraday 5-min data inserted.")
        else:
            print(f"No intraday data for {ticker}")

        latest_row = intraday.tail(1) if not intraday.empty else historical.tail(1)
        print(f"\n--- {ticker} Latest Row ---")
        print(latest_row)
        print(f"{ticker} fetch complete.\n")
```

```
def get_prices(ticker, limit=80):
    connection = get_connection()
    cursor = connection.cursor(dictionary=True)
    cursor.execute(
        "SELECT * FROM price_data WHERE ticker = %s ORDER BY timestamp",
        (ticker, limit))
    rows = cursor.fetchall()
    cursor.close()
    connection.close()
    return list(reversed(rows))

@app.route("/api/prices/<ticker>")
def prices_api(ticker):
    limit = request.args.get("limit", default=80, type=int)
    rows = get_prices(ticker, limit)
    return jsonify(rows)

def start_price_loop():
    sys.stdout.reconfigure(line_buffering=True)
    first_cycle = True

    while True:
        print("\n --- Retrieving Price Data ---", flush=True)
        fetch_and_store()
        print("Price Data Retrieved.\n", flush=True)

        if first_cycle:
            first_cycle = False
            print("Initial fetch complete. Next fetch will be in 5 minutes", flush=True)

        time.sleep(300)
```

Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 5001
CMD ["python", "-u", "app.py"]
```

requirements.txt

```
yfinance
flask
requests
pandas
mysql-connector-python
```