Fund-Screener Intern Screening Project

Fund-Screener Intern Project: Building a Production-Grade Financial Analysis Pipeline

Welcome to the Team! 🚀

This project mirrors real-world work as an AI Software Engineer. Your mission: build a **production-grade financial analysis pipeline** that demonstrates your ability to handle complex data engineering challenges.

Focus on **architectural decisions, error handling, and code quality**. We want to see how you think through problems, not just if you can write code.

---

1. Project Overview

What You're Building

A **command-line financial analysis factory** that:

1. **Ingests** daily price data and fundamental metrics from yfinance API
2. **Validates** raw data using Pydantic schemas
3. **Processes** and merges different data frequencies (daily vs quarterly)
4. **Calculates** technical indicators and fundamental ratios
5. **Detects** trading signals (Golden Crossover pattern)
6. **Persists** results to SQLite with proper schema design
7. **Delivers** analysis via CLI with JSON export

Skills You'll Demonstrate

- **Data Engineering**: ETL pipeline design, missing data strategies
- **Software Architecture**: Modular design, error handling
- **Database Design**: Schema design, idempotent operations
- **Production Practices**: Logging, testing, documentation

---

2. Key Technical Challenges (Examples)

Challenge 1: The Frequency Mismatch Problem

The Problem: Stock prices update daily, but financial statements only update quarterly.

Your Task:

- Design a merging strategy that makes financial sense.
- Handle days between quarterly reports.
- Document your reasoning and trade-offs in the README.

Challenge 2: Unreliable Fundamental Data

The Reality: `yfinance` fundamental data is often missing or incomplete.

Your Task:

- Implement a fallback strategy (alternative sources, interpolation, synthetic metrics).
- Document your decisions in README.md.
- Optional: Use other APIs like AlphaVantage, clearly noting the source.

Challenge 3: Golden Crossover Detection

The Logic: Detect when the 50-day SMA crosses above the 200-day SMA.

Your Task:

- Implement detection logic using vectorised pandas operations.
- Handle edge cases (insufficient data, NaN values).
- Return a list of crossover dates.

- Bonus: Implement Death Cross detection.

Challenge 4: Multi-Market & Recent Stock Handling

The Reality: Your pipeline must handle **all kinds of stocks**:

- Regular stocks with long histories (India & US)
- Recent listings no older than 10 months (e.g., Hyundai, Swiggy, Urban Company)
- Stocks from both Indian and US markets

Your Task:

- Ensure your pipeline works **seamlessly across all these cases**.
- Handle ticker differences between markets (e.g., `RELIANCE.NS` for India vs `NVDA` for the US).
- Gracefully handle:
  - Limited price history for recent listings
  - Missing or partial fundamental data
- Document your approach for ticker handling strategies.

Testing Requirement:

- Test the pipeline on **a mix of stocks**: old/regular US & Indian stocks, as well as recent listings (<10 months) from both markets.
- Your code should automatically adapt to market-specific tickers and varying data availability.

---

3. Development Setup

Prerequisites

- Python 3.9+
- `uv` (recommended) or `poetry`

Quick Start

```
# Using uv (recommended)
uv init financial_analyzer
cd financial_analyzer
uv add pandas yfinance pydantic "typer[all]" sqlalchemy pyyaml
uv add --dev ruff pytest

# Using poetry
poetry new financial_analyzer
cd financial_analyzer
poetry add pandas yfinance pydantic "typer[all]" sqlalchemy pyyaml
poetry add --group dev ruff pytest
```

Required Project Structure

```
financial_analyzer/
├── src/
│   ├── __init__.py
│   ├── data_fetcher.py      # API calls & validation
│   ├── processor.py         # Data merging & metrics
│   ├── signals.py           # Signal detection
│   ├── database.py          # SQLite operations
│   ├── models.py            # Pydantic schemas
│   ├── main.py              # CLI entry point
│   └── config.py            # Configuration
├── tests/
│   ├── test_processor.py    # Test calculations
│   ├── test_signals.py      # Test signal detection
│   └── conftest.py
├── config.yaml.example
├── pyproject.toml
└── README.md
```

---

4. Module Requirements

`src/models.py` - Pydantic Schemas

- Validate raw API responses (price data, fundamentals).
- Validate processed metrics (daily calculations).
- Validate signal events (crossovers with metadata).
- Validate the final JSON export format.
- Enforce price relationships (High ≥ Low, etc.).
- Handle optional fundamental data gracefully.
- Use appropriate data types (`Decimal` for currency).

`src/data_fetcher.py` - Data Ingestion

Core Function: `fetch_stock_data(ticker: str) → dict`

- Fetch 5 years of daily OHLCV data (or as available for recent IPOs).
- Implement a fallback for missing fundamental data.
- Validate all responses with Pydantic models.
- Handle API timeouts and errors gracefully.
- Log data quality issues.

Fundamental Data Strategy:

1. Try `ticker.quarterly_balance_sheet`
2. Fall back to `ticker.balance_sheet` (annual)
3. Use `ticker.info` for basic metrics
4. Document source used

`src/processor.py` - Data Pipeline

Core Function: `process_data(raw_data: dict) → pd.DataFrame`

- Merge daily prices with quarterly fundamentals.
- Handle missing fundamental data with forward-fill.
- Calculate technical indicators:

    - 50-day and 200-day SMA
    - 52-week high and % difference from high
- Calculate fundamental ratios:
    - Book Value per Share
    - Price-to-Book Ratio
    - Enterprise Value (simplified)
- Document why forward-fill is reasonable for fundamentals.

`src/signals.py` - Signal Detection

Core Function: `detect_golden_crossover(df: pd.DataFrame) -> List[date]`

- Detect Golden Crossover with vectorised operations.
- Handle edge cases (insufficient data, NaN values).
- Return a list of crossover dates.
- Bonus: Death Cross detection.

`src/database.py` - SQLite Operations

Required Tables:

- `tickers` – Basic stock info
- `daily_metrics` - All calculated metrics
- `signal_events` – Detected trading signals

Requirements:

- Use SQLAlchemy ORM.
- UNIQUE constraints to prevent duplicates.
- Idempotent insert operations (INSERT OR REPLACE).
- Functions to save DataFrames to tables.

`src/main.py` - CLI Interface

- Built with Typer.

- Accept `-ticker` and `-output` arguments.
- Orchestrate full pipeline:
    1. Initialize database
    2. Fetch and validate data
    3. Process and calculate metrics
    4. Detect signals
    5. Save to database and JSON
    6. Log success/failure

Example Usage:

```
# US stock
python -m src.main --ticker NVDA --output nvda_analysis.json

# Indian stock
python -m src.main --ticker RELIANCE.NS --output reliance_analysis.json

# Recent IPO (India or US)
python -m src.main --ticker SWIGGY.NS --output swiggy_analysis.json
```

---

5. Configuration & Error Handling

`config.yaml.example`

```
database:
  path: "financial_data.db"

logging:
  level: "INFO"

data_settings:
```

```
historical_period: "5y"
min_trading_days_for_sma: 200
```

Error Handling Strategy

- Use **logging**, not print statements.
- Handle API failures gracefully.
- Continue processing with partial data when possible.
- Document data quality issues in output.
- Handle missing or partial fundamentals (forward-fill or synthetic values).
- Ensure the pipeline works even for short histories (recent IPOs).

---

6. Production Standards (Mandatory)

Code Quality

- **Type hints**: Required for all functions.
- **Docstrings**: Google-style for all public functions.
- **Linting**: Run `ruff check . --fix && ruff format .`.

Testing (Optional)

- Write pytest tests for:
  - Metric calculations (verify SMA math).
  - Signal detection logic (test crossover detection).
  - Data validation (Pydantic model tests).
  - Test the pipeline on **old and recent stocks from Indian & US markets**.

Database Design

- UNIQUE constraints on `(ticker, date)` combinations.
- Idempotent operations (INSERT OR REPLACE).

- Document the schema in the README.

---

7. Deliverables

GitHub Repository (Public)

Your repo must include:

- **Source Code**: All modules (`src/`) following the required structure.
- **Configuration**: `config.yaml.example`
- **Tests**: Pytest test cases (`tests/`).
- **Output Data**: JSON files generated by your CLI (`-output`) for all tickers you tested (old & recent, India & US).
- **Documentation**: Complete `README.md` including:
  1. Project Overview
  2. Setup Instructions
  3. Usage Examples
  4. Database Schema
  5. Design Decisions (forward-fill, missing data, idempotency, ticker handling)
  6. Data Quality Notes
  7. Testing Instructions (include old and recent stocks from India & US)

Note: Including the JSON outputs ensures we can validate your pipeline runs correctly across different stock types.

---

8. Submission & Presentation

- Submit the **GitHub link**.
- If shortlisted, you must:
  - Prepare a **presentation** explaining your decisions.
  - Walk through your pipeline, schema, and metrics.

- Answer **cross-questions** about your implementation choices and related concepts.
- Some things that can be asked:
  - Architecture decisions
    - Data quality handling
    - Database design choices
    - Error handling strategy
    - Cross-market & IPO-age handling

---

9. Getting Help

Common Pitfalls

- Don’t give up if fundamental data is missing.
- Don’t assume all data is clean.
- Don’t ignore edge cases in signal detection.
- Don’t skip documentation.
- Don’t hardcode ticker formats.

Questions to Ask Yourself

- How do I handle missing fundamental data?
- What if there’s insufficient price history for SMA?
- How do I prevent duplicates in the database?
- How do I adapt tickers across India & US markets?
- How do I handle recent IPOs (<10 months)?

---

10. Quick-Start & Testing Cheat Sheet

1. CLI Usage Examples

US Stocks

```
python -m src.main --ticker NVDA --output nvda_analysis.json
python -m src.main --ticker AAPL --output aapl_analysis.json
```

Indian Stocks

```
python -m src.main --ticker RELIANCE.NS --output reliance_analysis.json
python -m src.main --ticker TCS.NS --output tcs_analysis.json
```

Recent IPOs/Listings (<10 months)

```
python -m src.main --ticker SWIGGY.NS --output swiggy_analysis.json
python -m src.main --ticker HYUNDAI.NS --output hyundai_analysis.json
python -m src.main --ticker URBANCOMP.NS --output urbancomp_analysis.js
on
```

---

2. Testing Checklist

✅ **Stock Types**

- Old/regular US stocks
- Old/regular Indian stocks
- Recent IPOs (<10 months) US & Indian stocks

✅ **Pipeline Functions**

- Fetch & validate data (OHLCV & fundamentals)
- Merge daily & quarterly data
- Calculate technical indicators: 50/200-day SMA, 52-week high
- Calculate fundamental ratios: P/B, BVPS, EV
- Detect Golden Crossover / Death Cross
- Save to SQLite database (idempotent insert)
- Export JSON results

✅ **Edge Cases**

- Missing fundamental data
- Short price history (recent IPOs)
- NaN or partial data in calculations
- Cross-market tickers handled correctly

---

3. Logging & Debugging

- Logging level in `config.yaml`:

```
logging:
  level: "INFO"
```

- Check logs for:
  - Missing data fallback usage
  - API errors or timeouts
  - Idempotent database insert messages

---

4. Recommended Tickers for Testing

Market	Stock Type	Example Tickers
US	Old / Regular	NVDA, AAPL, MSFT
US	Recent IPO	(any <10 months)
India	Old / Regular	RELIANCE.NS, TCS.NS
India	Recent IPO	SWIGGY.NS, HYUNDAI.NS, URBANCOMP.NS

---

5. Quick Tips

- Always **validate data** before processing.
- Use **forward-fill** or synthetic metrics for missing fundamentals.
- Ensure **database inserts are idempotent** (no duplicates).

- Handle **market-specific ticker formats** (`.NS` for NSE, no suffix for the US markets).
- Document your **design decisions and assumptions** in README.md.

Good luck! We're excited to see your approach to these real-world data engineering challenges.