Stock Data Collection System - LLM Development Guide

Table of Contents

System Overview

Business Requirements

Build a Python application to collect and manage financial data for 2,000+ stocks with the following capabilities:

- **Daily price updates** for all tracked stocks (OHLCV data)
- **Historical data collection** for new stocks (200 days back)
- **Company fundamental data** (financial metrics, ratios, company info)
- **Analyst estimates and earnings data** (earnings forecasts, analyst ratings)
- **Technical indicators calculation** (200-day MA, 20-day RSI, Bollinger Bands)

Data Sources

- **Primary**: Alpha Vantage API (free tier: 25 calls/day, 5 calls/minute)
- **Secondary**: Yahoo Finance via yfinance library (unofficial, no hard limits)

Key Success Metrics

- Process 2,000 stocks daily within API limits
- Maintain 99%+ data coverage
- Complete daily updates in 1-2 hours (conservative approach to avoid blocking)
- Stay within free tier API allocations
- Return detailed JSON status via REST API endpoint

API Analysis and Strategy

Alpha Vantage API Capabilities

1. Bulk Quotes Endpoint (REALTIME_BULK_QUOTES)

```
Function: REALTIME_BULK_QUOTES
Capacity: 100 stocks per API call
Daily Allocation: 20 calls = 2,000 stocks
Execution Time: 1-2 hours with conservative rate limiting (15-20 second delays)
Use Case: Primary method for daily price updates
```

2. Historical Data Endpoint (TIME_SERIES_DAILY)

```
Function: TIME_SERIES_DAILY
Capacity: 1 stock per API call
Daily Allocation: 3-5 calls = 3-5 new stocks daily
Data Range: Full history available (outputsize=full)
Use Case: Onboarding new stocks with historical data
```

3. Company Fundamentals Endpoint (OVERVIEW)

```
Function: OVERVIEW
Capacity: 1 company per API call
Daily Allocation: 1-2 calls = 7-14 companies weekly
Data Returned: Market cap, P/E, EPS, revenue, margins, etc.
Use Case: Fundamental analysis and screening
```

4. Earnings Data Endpoint (EARNINGS)

```
Function: EARNINGS
Capacity: 1 company per API call
Daily Allocation: 1-2 calls = 7-14 companies weekly
Data Returned: Quarterly/annual earnings, analyst estimates
Use Case: Earnings analysis and forecast tracking
```

Yahoo Finance Fallback Strategy

```
Library: yfinance
Batch Capacity: 50-100 stocks per request (recommended)
Rate Limits: No official limits, but conservative batching required
Use Case: Backup for Alpha Vantage failures, current price data
Threading: Enable multithreading for faster batch processing
```

Configuration Management

LLM PROMPT: Configuration System Implementation

Create a configuration management system with the following specifications:

1. **JSON Configuration File Structure**:

json

```
{
  "alpha_vantage": {
    "api_key": "PLACEHOLDER_FOR_USER_KEY",
    "daily_limit": 25,
    "rate_limit_per_minute": 5,
    "wait_between_calls": 15,
    "conservative_mode": true,
    "endpoints": {
      "bulk_quotes": {
        "function": "REALTIME_BULK_QUOTES",
        "stocks_per_call": 100,
        "daily_allocation": 20,
        "wait_time": 15,
        "priority": 1
      },
      "historical_data": {
        "function": "TIME_SERIES_DAILY",
        "stocks_per_call": 1,
        "daily_allocation": 3,
        "wait_time": 20,
        "priority": 2
      },
      "company_overview": {
        "function": "OVERVIEW",
        "stocks_per_call": 1,
        "daily_allocation": 1,
        "wait_time": 20,
        "priority": 3
      },
      "earnings_data": {
        "function": "EARNINGS",
        "stocks_per_call": 1,
        "daily_allocation": 1,
        "wait_time": 20,
        "priority": 4
      }
    }
  },
  "yahoo_finance": {
    "batch_size": 25,
    "wait_between_batches": 10,
    "enable_threading": false,
    "conservative_mode": true
  },
  "web_service": {
    "host": "0.0.0.0",
```

```
    "port": 8000,
    "endpoint": "/collect_stock_data"
  }
}
```

2. **Rate Limit Manager Class Requirements**:
   - Track daily API usage by endpoint
   - Enforce daily and per-minute limits
   - Reset usage tracking at midnight
   - Provide methods: `can_make_call(endpoint)`, `record_call(endpoint)`, `get_remaining_calls()`
   - Print usage status with remaining allocations
3. **Configuration Validation**:
   - Verify total daily allocations don't exceed API limits
   - Validate endpoint configurations match Alpha Vantage API specs
   - Check that API key is provided and valid format

Core Implementation Requirements

LLM PROMPT: API Client Implementation

Implement the following API client classes with specific error handling and retry logic:

1. Alpha Vantage Client Class

python

```python
class AlphaVantageClient:
    def __init__(self, api_key: str, rate_limiter: RateLimitManager):
        # Initialize with API key and rate limiter reference

    def get_bulk_quotes(self, symbols: List[str]) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Check rate limits before making call
        # - Format symbols as comma-separated string (max 100)
        # - Handle API errors: "Error Message", "Note" in response
        # - Return standardized format: {'successful': int, 'failed': int, 'data': List}
        # - Record API call in rate limiter
        # - Implement timeout of 30 seconds

    def get_historical_data(self, symbol: str) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Use outputsize='full' for complete history
        # - Parse "Time Series (Daily)" from response
        # - Convert to standardized OHLCV format
        # - Handle missing data gracefully
        # - Return format: {'symbol': str, 'data': Dict, 'success': bool}

    def get_company_overview(self, symbol: str) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Extract key fundamental metrics: MarketCapitalization, PE, EPS, etc.
        # - Handle cases where fundamental data is "None" or missing
        # - Return format: {'symbol': str, 'fundamentals': Dict, 'success': bool}

    def get_earnings_data(self, symbol: str) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Parse both quarterly and annual earnings
        # - Extract analyst estimates if available
        # - Handle earnings date formatting
        # - Return format: {'symbol': str, 'earnings': Dict, 'success': bool}
```

2. Yahoo Finance Client Class

python

```python
class YahooFinanceClient:
    def __init__(self, config: Dict):
        # Initialize with Yahoo Finance configuration

    def get_batch_quotes(self, symbols: List[str]) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Use yfinance.download() with threading enabled
        # - Process in batches based on config batch_size
        # - Handle multi-level column structure from yfinance
        # - Extract current price, volume, and basic OHLC
        # - Return format compatible with Alpha Vantage responses
        # - Implement retry logic for failed batches
```

LLM PROMPT: Data Processing Functions

Create data processing functions with the following specifications:

python

```python
def standardize_price_data(raw_data: Dict, source: str) -> List[Dict]:
    # IMPLEMENTATION REQUIREMENTS:
    # - Convert both Alpha Vantage and Yahoo Finance formats to standard format
    # - Standard format: [{'symbol': str, 'date': str, 'open': float, 'high': float, 'low': flc
    # - Handle missing values by setting to None
    # - Validate data types and ranges (prices > 0, volume >= 0)
    # - Add source field to track data origin

def calculate_technical_indicators(price_history: List[Dict]) -> Dict:
    # IMPLEMENTATION REQUIREMENTS:
    # - Implement 200-day Simple Moving Average
    # - Implement 20-day RSI calculation
    # - Implement Bollinger Bands (20-day, 2 standard deviations)
    # - Handle insufficient data gracefully (less than required periods)
    # - Return format: {'sma_200': float, 'rsi_20': float, 'bb_upper': float, 'bb_lower': float
    # - Use pandas for efficient calculations if available

def validate_stock_symbol(symbol: str) -> bool:
    # IMPLEMENTATION REQUIREMENTS:
    # - Check symbol format (alphanumeric, dots, dashes allowed)
    # - Maximum length validation (10 characters)
    # - Return True if valid, False otherwise
```

REST API Implementation

LLM PROMPT: Web Service Implementation

Create a Flask/FastAPI web service with the following specifications:

1. REST API Endpoint

python

```python
from flask import Flask, jsonify, request
from datetime import datetime
import threading

app = Flask(__name__)

@app.route('/collect_stock_data', methods=['GET'])
def collect_stock_data():
    """
    IMPLEMENTATION REQUIREMENTS:
    - Accept GET request to trigger data collection
    - Run data collection in background thread
    - Return immediate response with execution status
    - Provide detailed JSON response with specified format
    - Handle concurrent requests gracefully
    - Log all requests and responses
    """

    # Expected JSON Response Format:
    response_format = {
        "daily_prices": 0,           # Number of stocks updated with daily prices
        "historical_prices": 0,     # Number of stocks updated with historical data
        "alpha_calls": 0,           # Number of Alpha Vantage API calls made
        "yahoo_calls": 0,           # Number of Yahoo Finance API calls made
        "new_tickers": [],          # List of new tickers added
        "failed": [],               # List of tickers that failed processing
        "errors": [],               # List of error messages
        "execution_time": 0,        # Total execution time in seconds
        "start_time": "",           # ISO format timestamp
        "end_time": "",             # ISO format timestamp
        "status": "completed"       # "running", "completed", "failed"
    }
```

2. Background Processing

python

```python
def background_collection_task():
    """
    IMPLEMENTATION REQUIREMENTS:
    - Run complete data collection process
    - Update global status variables during execution
    - Handle all errors and exceptions
    - Store results in database
    - Update response JSON throughout process
    - Implement proper logging
    """

def get_collection_status():
    """
    IMPLEMENTATION REQUIREMENTS:
    - Return current execution status
    - Include real-time progress updates
    - Show partial results during execution
    - Handle multiple concurrent status requests
    """
```

3. Service Management

python

```python
# Server startup configuration
if __name__ == '__main__':
    app.run(
        host='0.0.0.0',
        port=8000,
        debug=False,
        threaded=True
    )

# Health check endpoint
@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({"status": "healthy", "timestamp": datetime.now().isoformat()})

# Status endpoint for monitoring
@app.route('/status', methods=['GET'])
def get_status():
    return jsonify(get_collection_status())
```

Database Integration

LLM PROMPT: Database Schema and Operations

Implement database operations with the following schema and requirements:

1. Database Schema (PostgreSQL daily_charts table)

```sql
-- daily_charts table (existing schema)
CREATE TABLE daily_charts (
    id integer DEFAULT nextval('daily_charts_id_seq'::regclass) PRIMARY KEY,
    ticker character varying(20) NOT NULL,
    date character varying(20) NOT NULL,
    open numeric(7,2),
    high numeric(7,2),
    low numeric(7,2),
    close numeric(7,2),
    volume integer,
    ema_20 numeric(7,2),
    ema_50 numeric(7,2),
    ema_100 numeric(7,2),
    cci numeric(8,2),
    rsi numeric(5,2),
    bb_upper numeric(7,2),
    bb_middle numeric(7,2),
    bb_lower numeric(7,2)
);

-- Create index for efficient queries
CREATE INDEX idx_daily_charts_ticker_date ON daily_charts(ticker, date);
CREATE INDEX idx_daily_charts_date ON daily_charts(date);
```

2. Database Operations Class

python

```python
import psycopg2
from psycopg2.extras import RealDictCursor
import pandas as pd
from datetime import datetime, timedelta

class DailyChartsDatabase:
    def __init__(self, connection_string: str):
        # IMPLEMENTATION REQUIREMENTS:
        # - Initialize PostgreSQL connection with connection pooling
        # - Handle connection errors gracefully
        # - Use parameterized queries for security

    def store_daily_price_data(self, price_data: List[Dict]) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Insert/Update daily price data in daily_charts table
        # - Calculate and store technical indicators (EMAs, RSI, CCI, Bollinger Bands)
        # - Use UPSERT (INSERT ... ON CONFLICT UPDATE) for handling duplicates
        # - Return statistics: {'inserted': int, 'updated': int, 'failed': int}
        # - Handle data type conversions (ensure numeric precision)

    def calculate_and_store_technical_indicators(self, ticker: str) -> bool:
        # IMPLEMENTATION REQUIREMENTS:
        # - Calculate EMA_20, EMA_50, EMA_100 using exponential moving averages
        # - Calculate RSI using 14-period relative strength index
        # - Calculate CCI using 20-period commodity channel index
        # - Calculate Bollinger Bands (bb_upper, bb_middle, bb_lower) using 20-period SMA and 2
        # - Update existing records with calculated values
        # - Return success/failure status

    def get_tickers_needing_historical_data(self) -> List[str]:
        # IMPLEMENTATION REQUIREMENTS:
        # - Find tickers with less than 200 days of data
        # - Return list ordered by data availability (least data first)
        # - Exclude weekends and holidays from count

    def get_latest_date_for_ticker(self, ticker: str) -> str:
        # IMPLEMENTATION REQUIREMENTS:
        # - Return latest date available for specific ticker
        # - Handle case where ticker doesn't exist
        # - Return in YYYY-MM-DD format

    def get_all_tickers(self) -> List[str]:
        # IMPLEMENTATION REQUIREMENTS:
        # - Return distinct list of all tickers in database
        # - Order alphabetically
```

```python
    def validate_data_integrity(self) -> Dict:
        # IMPLEMENTATION REQUIREMENTS:
        # - Check for missing dates, invalid prices, volume anomalies
        # - Return validation report with issues found
        # - Format: {'valid_records': int, 'issues': List[str]}
```

Error Handling and Fallbacks

LLM PROMPT: Error Handling Implementation

Implement comprehensive error handling with the following requirements:

python

```python
class StockDataCollectionError(Exception):
    """Base exception for stock data collection errors"""
    pass

class RateLimitExceededError(StockDataCollectionError):
    """Raised when API rate limits are exceeded"""
    pass

class DataValidationError(StockDataCollectionError):
    """Raised when data validation fails"""
    pass

def handle_api_errors(func):
    """Decorator for API call error handling"""
    # IMPLEMENTATION REQUIREMENTS:
    # - Catch connection timeouts, HTTP errors
    # - Implement exponential backoff for retries (3 attempts)
    # - Log all errors with timestamp and context
    # - Return standardized error response format
    # - Don't retry on rate limit errors (switch to fallback instead)

def fallback_strategy(primary_failed: str, symbols: List[str]) -> Dict:
    # IMPLEMENTATION REQUIREMENTS:
    # - If Alpha Vantage bulk quotes fail -> use Yahoo Finance
    # - If Alpha Vantage historical fails -> queue for next day
    # - If Yahoo Finance fails -> log and return partial results
    # - Track fallback usage for monitoring
    # - Return results in same format as primary method
```

Daily Execution Strategy

LLM PROMPT: Main Execution Function

Create the main daily execution function with the following logic:

python

```
def execute_stock_data_collection() -> Dict:
    """
    Main execution function for REST API

    IMPLEMENTATION REQUIREMENTS:
    1. Initialize configuration and rate limiter
    2. Load ticker list from daily_charts database
    3. Execute tasks in priority order with conservative timing:
       - Priority 1: Current price updates (all stocks) - 15-20 second delays
       - Priority 2: Historical data (new stocks) - 20 second delays
       - Priority 3: Company fundamentals (rotation) - 20 second delays
       - Priority 4: Earnings data (rotation) - 20 second delays
    4. Handle fallbacks gracefully with extended timeouts
    5. Calculate and store technical indicators for updated stocks
    6. Update database with collected data immediately after each successful API call
    7. Return detailed JSON response matching specified format
    8. Allow 1-2 hours total execution time

    EXECUTION FLOW:
    - Initialize response JSON with default values
    - Track start time
    - Load all tickers from daily_charts table
    - For each task type:
      * Check if sufficient API calls remain
      * Execute with conservative error handling and longer delays
      * Update database immediately after each successful call
      * Update response JSON with progress
      * Log detailed progress and results
    - Calculate technical indicators for all updated tickers
    - Update end time and execution statistics
    - Return complete response JSON

    RESPONSE FORMAT:
    {
        "daily_prices": 2000,           # Number of stocks updated with daily prices
        "historical_prices": 3,         # Number of stocks updated with historical data
        "alpha_calls": 24,              # Number of Alpha Vantage API calls made
        "yahoo_calls": 5,               # Number of Yahoo Finance API calls made
        "new_tickers": ["NVDA", "AMD"], # List of new tickers added
        "failed": ["BADTICK"],          # List of tickers that failed processing
        "errors": ["API rate limit"],   # List of error messages
        "execution_time": 3600,         # Total execution time in seconds
        "start_time": "2025-06-02T09:00:00",
        "end_time": "2025-06-02T10:00:00",
        "status": "completed"           # "running", "completed", "failed"
    }
```

```
    """

def get_tickers_for_processing() -> Dict:
    """
    IMPLEMENTATION REQUIREMENTS:
    - Get all unique tickers from daily_charts table
    - Identify tickers needing historical data (less than 200 trading days)
    - Select tickers for fundamental updates (weekly rotation)
    - Select tickers for earnings updates (weekly rotation)
    - Return categorized ticker lists
    """

def conservative_rate_limiting(last_call_time: float, wait_seconds: int):
    """
    IMPLEMENTATION REQUIREMENTS:
    - Implement conservative rate limiting with jitter
    - Add random delay of 1-3 seconds to avoid patterns
    - Ensure minimum wait time between calls
    - Log delay information for monitoring
    """

def update_response_progress(response: Dict, task: str, progress: Dict):
    """
    IMPLEMENTATION REQUIREMENTS:
    - Update response JSON during execution
    - Track real-time progress
    - Handle concurrent access safely
    - Maintain response format consistency
    """
```

Code Structure Specifications

LLM PROMPT: Project Structure and Organization

Create the following project structure with specified functionality:

```
stock_data_collector/
├── config/
│   ├── config.json                 # Main configuration
│   ├── rate_limits.json           # Rate limit settings
│   └── database_config.json       # PostgreSQL connection settings
├── src/
│   ├── __init__.py
│   ├── web_service/
│   │   ├── __init__.py
│   │   ├── app.py                  # Flask/FastAPI main application
│   │   ├── routes.py               # REST API endpoints
│   │   └── background_tasks.py     # Background data collection
│   ├── api_clients/
│   │   ├── __init__.py
│   │   ├── alpha_vantage_client.py # Alpha Vantage API wrapper
│   │   ├── yahoo_finance_client.py # Yahoo Finance wrapper
│   │   └── rate_limiter.py         # Rate limiting logic
│   ├── data_processing/
│   │   ├── __init__.py
│   │   ├── data_standardizer.py    # Format standardization
│   │   ├── technical_indicators.py # Technical analysis calculations
│   │   └── data_validator.py       # Data quality validation
│   ├── database/
│   │   ├── __init__.py
│   │   ├── connection.py           # PostgreSQL connection management
│   │   ├── daily_charts_ops.py     # daily_charts table operations
│   │   └── technical_calc.py       # Technical indicators calculation
│   └── utils/
│       ├── __init__.py
│       ├── logging_config.py       # Logging setup
│       ├── error_handlers.py       # Error handling utilities
│       └── helpers.py              # General utility functions
├── scripts/
│   ├── setup_database.py          # Database initialization
│   ├── config_generator.py        # Generate configuration files
│   └── test_api_endpoints.py      # Test REST API endpoints
├── tests/
│   ├── test_api_clients.py
│   ├── test_data_processing.py
│   ├── test_database_operations.py
│   └── test_web_service.py
├── requirements.txt
├── README.md
└── main.py                         # Web service entry point
```

Key Implementation Files

requirements.txt

```
requests>=2.28.0
yfinance>=0.2.18
pandas>=1.5.0
numpy>=1.24.0
sqlalchemy>=1.4.0
psycopg2-binary>=2.9.0  # for PostgreSQL
python-dotenv>=0.19.0
pyyaml>=6.0
flask>=2.3.0
flask-cors>=4.0.0
gunicorn>=20.1.0  # for production deployment
```

main.py

python

```python
#!/usr/bin/env python3
"""
Stock Data Collection System - REST API Web Service

IMPLEMENTATION REQUIREMENTS:
- Flask web service with REST API endpoints
- Background data collection processing
- Comprehensive error handling and logging
- JSON response format as specified
- Conservative rate limiting for 1-2 hour execution

Usage examples:
- python main.py                    # Start web service
- GET /collect_stock_data          # Trigger data collection
- GET /status                      # Check collection status
- GET /health                      # Health check endpoint
"""
```

Testing and Validation Requirements

LLM PROMPT: Testing Implementation

Create comprehensive tests with the following coverage:

```python
# Test Categories Required:

1. API Client Tests:
   - Mock API responses for Alpha Vantage endpoints
   - Test rate limiting enforcement
   - Test error handling and retries
   - Test data format standardization

2. Data Processing Tests:
   - Test technical indicator calculations with known datasets
   - Test data validation rules
   - Test handling of missing/invalid data

3. Database Operation Tests:
   - Test CRUD operations with test database
   - Test batch insertion performance
   - Test data integrity constraints

4. Integration Tests:
   - Test complete daily collection workflow
   - Test fallback strategies
   - Test configuration loading and validation

5. Performance Tests:
   - Test execution time for 2000 stocks
   - Test memory usage during batch processing
   - Test database query performance
```

Deployment and Monitoring

LLM PROMPT: Production Deployment Setup

Create deployment configuration with monitoring:

python

```python
# Logging Configuration
logging_config = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
    },
    'handlers': {
        'file': {
            'level': 'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': 'stock_collector.log',
            'maxBytes': 10485760,  # 10MB
            'backupCount': 5,
            'formatter': 'standard',
        },
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'standard',
        },
    },
    'loggers': {
        '': {  # root logger
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': False
        }
    }
}

# Monitoring Requirements:
# - REST API request/response tracking
# - API usage monitoring and alerting
# - Data quality metrics (missing data, outliers)
# - Performance metrics (execution time, memory usage)
# - Error rate tracking and alerting
# - Background task status monitoring
```

Web Service Deployment

bash

```bash
# Production deployment with Gunicorn
gunicorn -w 4 -b 0.0.0.0:8000 main:app

# Docker deployment
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:8000", "main:app"]

# Environment variables
ALPHA_VANTAGE_API_KEY=your_key_here
DATABASE_URL=postgresql://user:pass@host:5432/dbname
FLASK_ENV=production
```

Health Monitoring

python

```python
# Health check endpoint implementation
@app.route('/health')
def health_check():
    return jsonify({
        "status": "healthy",
        "timestamp": datetime.now().isoformat(),
        "database": check_database_connection(),
        "api_quota": get_remaining_api_calls()
    })

# External monitoring integration
# - Add application performance monitoring (APM)
# - Set up alerts for API failures
# - Monitor response times and throughput
# - Track data collection success rates
```

Final Implementation Checklist

LLM PROMPT: Implementation Validation

Ensure the following functionality is fully implemented:

- [ ] Configuration management with JSON loading and validation

- [ ] Rate limiting with daily and per-minute enforcement (conservative 15-20 second delays)
- [ ] Alpha Vantage client with all 4 endpoints (bulk quotes, historical, overview, earnings)
- [ ] Yahoo Finance fallback client with conservative batch processing
- [ ] PostgreSQL daily_charts table integration with UPSERT operations
- [ ] Technical indicators calculation (EMA_20, EMA_50, EMA_100, RSI, CCI, Bollinger Bands)
- [ ] REST API web service with Flask/FastAPI
- [ ] Background task processing with status tracking
- [ ] JSON response format matching specifications
- [ ] Comprehensive error handling with fallback strategies
- [ ] Real-time progress updates during 1-2 hour execution
- [ ] Logging and monitoring with file rotation
- [ ] Health check and status endpoints
- [ ] Unit tests covering critical functionality
- [ ] Conservative rate limiting optimization to prevent blocking
- [ ] External triggering capability via GET requests

Success Criteria Validation

- [ ] Can process 2000 stocks in 1-2 hours with conservative rate limiting
- [ ] Stays within Alpha Vantage free tier limits (25 calls/day)
- [ ] Handles API failures gracefully with fallbacks
- [ ] Maintains 99%+ daily data coverage
- [ ] Provides comprehensive logging and error reporting
- [ ] Returns specified JSON response format via REST API
- [ ] Can onboard new stocks with historical data
- [ ] Calculates and stores all technical indicators (EMAs, RSI, CCI, Bollinger Bands)
- [ ] Integrates with existing daily_charts PostgreSQL table
- [ ] Provides real-time status updates during execution
- [ ] Implements conservative rate limiting to avoid blocking
- [ ] Supports external triggering via GET endpoint

This document provides complete specifications for implementing a production-ready stock data collection system. Follow the implementation requirements exactly as specified, paying particular attention to error handling, rate limiting, and data validation to ensure reliable operation within API constraints.