FINMENTOR AI - MULTIAGENT FINANCIAL ADVISORY PLATFORM

SOFTWARE DESIGN DOCUMENTATION

Submitted by

2448046: Roshan Varghese

2448059: Varun Alfred Dsouza

Project Guide: Dr. Saleema J S

For the course

Course Code: MDS581A

Course Name: PROJECT-II

TABLE OF CONTENTS

1. INTRODUCTION

1.1 SYSTEM OVERVIEW

FinMentor AI is an intelligent financial advisory platform that combines:

a) Multi-Agent AI architecture: Specialized agents for different financial tasks.
b) Agentic RAG (Retrival-Augmented Generation).
c) Real-time market data: Integration with Yahoo Finance API.
d) Educational Content: 540+ curated financial terms and concepts.
e) Portfolio management: Risk-based investment recommendations.

The main goal of FinMentor AI is to cater to new investors in capital markets is

1) Financial education: Explain concepts adaptively.
2) Market analysis: Real-time stock data, fundamentals, and trends.
3) Portfolio builder: Generate a diversified portfolio based on risk tolerance.
4) Conversational AI: Natural language query processing.
5) Context awareness: Remember and learn from past conversations.

The core target users are:

1) Indian retail investors.
2) Beginners to intermediate financial knowledge.
3) Individuals seeking affordable financial guidance.

1.2 DESIGN OBJECTIVES

Primary objectives include

1. Accessibility
    a. Providing 24/7 financial advisory services.
    b. Zero-cost to minimal cost solution.
    c. Simple, conversational interface.
2. Intelligence
    a. Specialized agents for different financial domains.
    b. Context-aware responses using RAG.
    c. Adaptive explanations based on user level.
3. Accuracy
    a. Real-time market data integration.
    b. Self-verification for critical financial advice.
    c. Evidence-based recommendation.
4. Scalability
    a. Modular architecture supporting additional agents.

    b. Extensible to handle more users and features.
    c. Database designed for growth.
5. Security
    a. Secure authentication.
    b. Password encryption.
    c. Data privacy protection.

2. SYSTEM ARCHITECTURE

2.1 ARCHITECTURE OVERVIEW

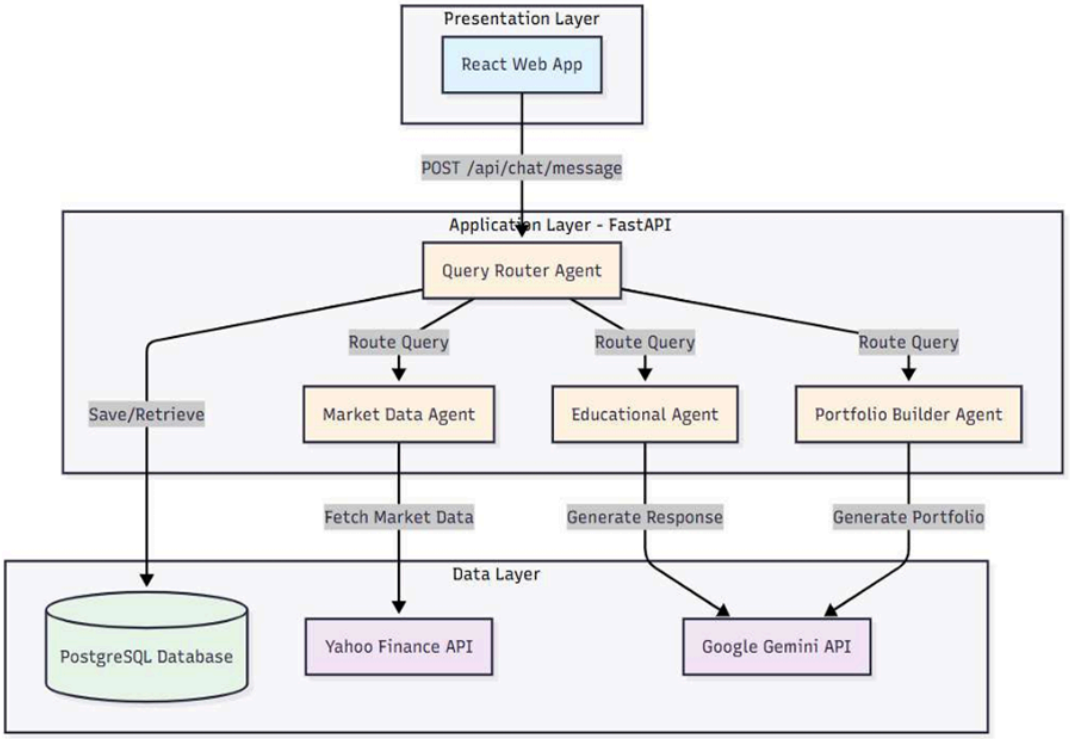
Finmentor AI follows a layered architecture pattern with clear separation of various modules. The system is designed as a modern,scalable multi-tier application.

1. Presentation layer:
    a. Web interface
    b. API consumers.
2. API gateway layer
    a. FastAPI framework
    b. Request routing.
    c. Authentication.
    d. Rate limiting.
3. Application layer
    a. Multi-agent system- DSPy+LangChain.
    b. Business logic
    c. Query processing.
    d. Response generation
4. Data access layer
    a. PostgreSQL database
    b. Vector search.
    c. ORM
    d. Caching.
5. Integration layer
    a. Yahoo finance API.
    b. Google gemini API.
    c. DuckDuckGo search.
    d. External data source (FINDER, Scraped data from ANGLE ONE)

Features:

- Modularity: Each component has single well defined responsibility.
- Scalability: Horizontal scaling through stateless services.

- Maintainability: Clear interface between layers.
- Testability: Components can be tested independently.
- Security: Defense in depth with multiple security layers.
- Performance: Caching and async processing where applicable.



2.2 MULTIAGENT ARCHITECTURE

The heart of FinMentor AI is its Multi-agent system that combines:

1. DSPy for structured reasoning.
2. LangChain for tool orchestration.
3. Agentic RAG for intelligent retrieval.

DSPy (Declarative Self-improving Python) provides the reasoning foundation.

2.2.1 DSPy REASONING LAYER

DSPy is a framework for LLM-based reasoning with structured signatures. It enables chain-of though reasoning and self optimizing through feedback. Each agent is defined as a DSPy signature.

Below is a brief anatomy of a DSPy agent.

1. Inputs (what the agent needs to reason):
    a. User query : str
    b. Context: str
    c. User profile: str
2. Reasoning process:
    a. Chain of thought
    b. Step-by-step analysis
    c. Evidence gathering
3. Outputs:
    a. Response: str
    b. Confidence: float
    c. Reasoning: str

The four core DSPy agents:

1. Query Router Agent:
    a. Purpose: Understand user intent and route to specialist.
    b. Signature: Query intention.
2. Educational Content Agent:
    a. Purpose: Explain financial concepts adaptively.
    b. Signature: Concept, explanation, examples
3. Market Data Agent:
    a. Purpose: Fetch and interpret market data.
    b. Signature: Stock symbol with price data, analysis and recommendation.
4. Portfolio builder agent:
    a. Purpose: Generate investment portfolios.
    b. Signature: Risk profile, goal, stock list.

2.2.2 LANGCHAIN ORCHESTRATION LAYER

Lanchain components in our system:

1. Tools (Actions agents can take):
    a. Get stock data: Fetch real Yahoo Finance data.
    b. Calculate portfolio metrics: Sharpe ratio, beta, etc..
    c. Search the financial web: DuckDuckGo search.
    d. Calculate technical indicators: RSI, moving averages.
2. Memory
    a. Conversational buffer window memory.
    b. Stores in PostgreSQL for persistence.
    c. Enables follow-up questions.
3. Agent executor

a. Run DSPy agents with tools
b. Handles errors and retries.

2.3 DESIGN RATIONALE

2.3.1 WHY MULTI-AGENT INSTEAD OF A SINGLE AGENT?

1. Accuracy: Specialists outperform generalists. An educational agent focuses only on teaching. The market agent focuses only on data.
2. Maintainability: It is easy to update individual agents, change portfolio logic. There is no risk of modifying the educational agent.
3. Performance: Multiple agents run simultaneously. Complex queries are answered faster.
4. Finally, we will get a better user experience.

2.3.2 WHY DSPY + LANGCHAIN HYBRID?

DECISION: Combine DSPy for reasoning + LangChain for tools

RATIONALE:

DSPy provides structured reasoning, chain of thought explanations, type-safe agent signatures, and self-improvement capabilities.

LangChain supports a rich ecosystem of tools, memory management, agent executors, and external API integrations.

2.3.4 Why PostgreSQL + Pgvector?

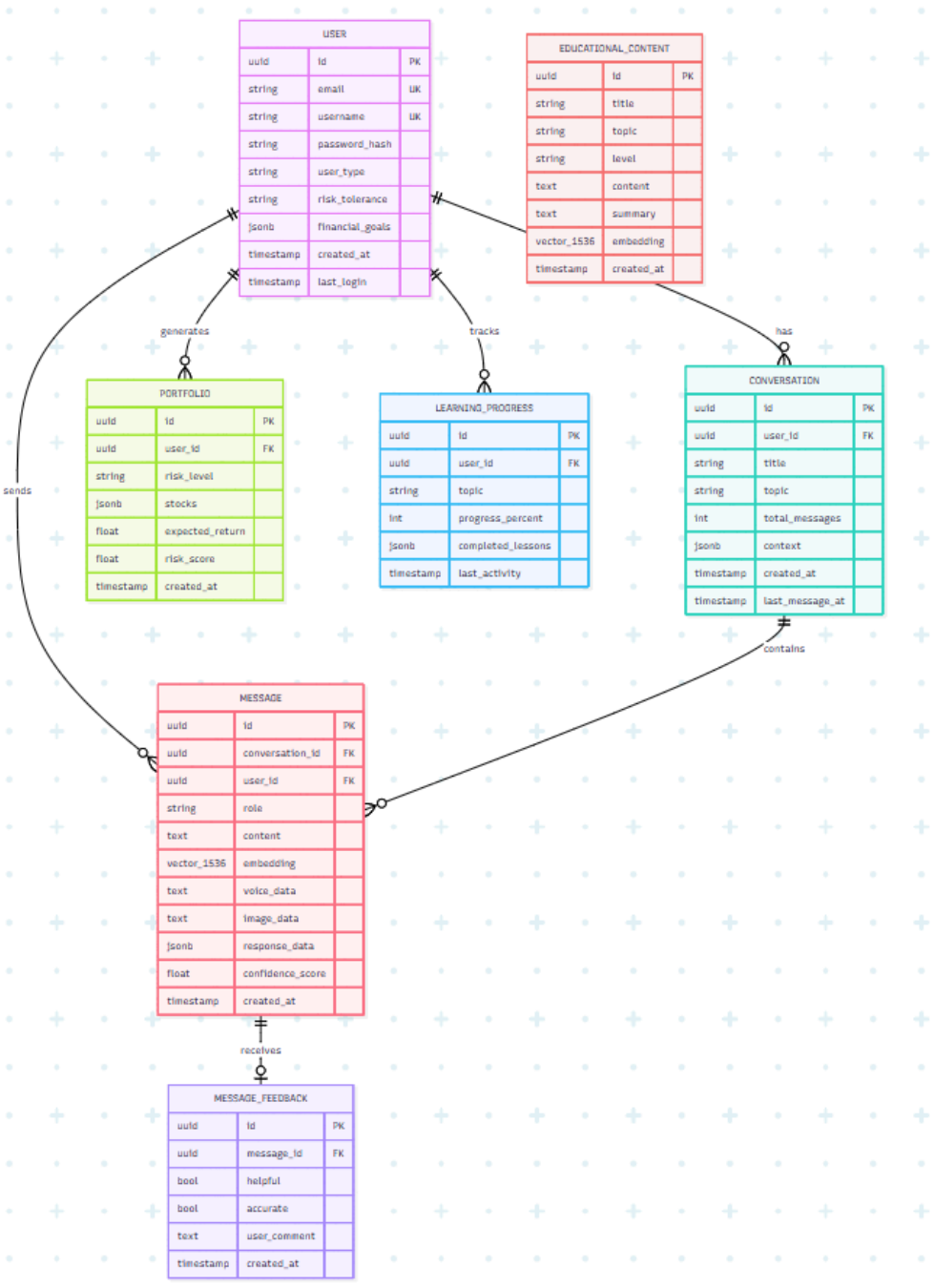
- PostgreSQL is a mature, reliable, and battle-tested database. It is ACID-compliant. Additionally, it is free and open-source.
- Pgvector is a native vector similarity search in Postgres. There is no need for a separate vector database.
- It can take in both SQL queries and vector search together.

3. DATABASE DESIGN

3.1 ENTITY RELATIONSHIP MODEL

1. User- Represents system users, stores profile, risk tolerance, and preferences.
2. Conversation- Represents a chat session and group-related messages together.
3. Message- Individual user query. This contains text+ vector embedding for RAG.

4. Portfolio- Generated investment recommendations. This is linked to the user and risk profile.
5. Educational content- Curated financial learning materials. Vector embeddings for similarity search.
6. Learning progress- Tracks users' financial literacy journey.



3.2 VECTOR EMBEDDINGS ARCHITECTURE

3.2.1 WHAT NEW VECTOR EMBEDDINGS

EMBEDDINGS transform text into mathematical vectors (arrays of numbers).

Example:

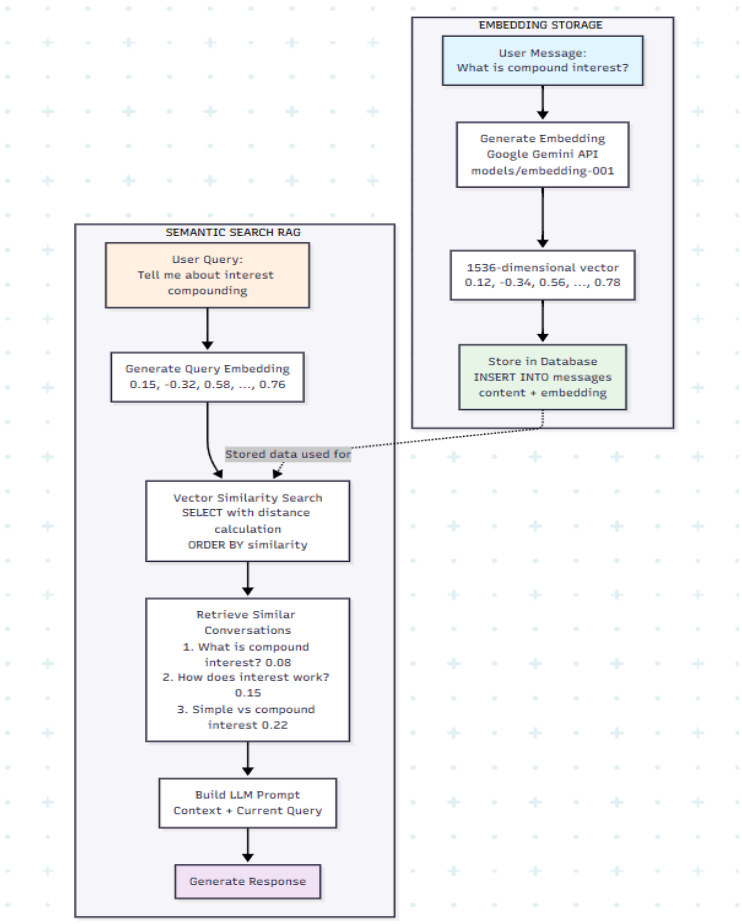
Text: "What is a mutual fund?"
(Embedding Model)
Vector: [0.23, -0.15, 0.87, ..., 0.34] // 1536 dimensions

SIMILAR MEANINGS → SIMILAR VECTORS

Text 1: "What is a mutual fund?"
Vector 1: [0.23, -0.15, 0.87, ...]
Text 2: "Explain mutual funds"
Vector 2: [0.25, -0.14, 0.88, ...] // Very close to Vector 1!
Text 3: "What is the stock market?"
Vector 3: [-0.65, 0.42, -0.21, ...] // Very different from Vector 1
DISTANCE CALCULATION:
distance(Vector1, Vector2) = 0.05 // Very similar!
distance(Vector1, Vector3) = 0.89 // Not similar

3.2.2 EMBEDDING GENERATION PIPELINE

Pipeline for message storage is shown below.



3.3 DATA INTEGRITY AND CONSTRAINTS

3.3.1 REFERENTIAL INTEGRITY

Cascade delete rules:

CASCADE DELETE Rules:

User deleted → Cascade delete:

- All conversations
- All messages
- All portfolios
- All learning progress

Conversation deleted → Cascade delete:

- All messages in that conversation

Message deleted → Cascade delete:

- Message feedback (if any)

FOREIGN KEY CONSTRAINTS:

```
ALTER TABLE conversations
ADD CONSTRAINT fk_user
FOREIGN KEY (user_id) REFERENCES users(id)
ON DELETE CASCADE;

ALTER TABLE messages
ADD CONSTRAINT fk_conversation
FOREIGN KEY (conversation_id) REFERENCES conversations(id)
ON DELETE CASCADE;
```

3.3.2 CHECK CONSTRAINTS

Ensure data validity at database level:

```
-- User age must be 18-100
ALTER TABLE users
ADD CONSTRAINT valid_age
CHECK (age >= 18 AND age <= 100);

-- Risk level must be valid
ALTER TABLE portfolios
ADD CONSTRAINT valid_risk
CHECK (risk_level IN ('low', 'medium', 'high'));

-- Confidence score must be 0-1
ALTER TABLE messages
ADD CONSTRAINT valid_confidence
CHECK (confidence_score >= 0 AND confidence_score <= 1);

-- Sentiment must be -1 to 1
```

```
ALTER TABLE conversations
ADD CONSTRAINT valid_sentiment
CHECK (sentiment >= -1 AND sentiment <= 1);
```

3.3.3 UNIQUE CONSTRAINTS

Prevent duplicates:

```
-- Email must be unique
ALTER TABLE users
ADD CONSTRAINT unique_email UNIQUE (email);
-- Username must be unique
ALTER TABLE users
ADD CONSTRAINT unique_username UNIQUE (username);
-- One feedback per message
ALTER TABLE message_feedback
ADD CONSTRAINT unique_message_feedback UNIQUE (message_id);
```

3.3.4 NOT NULL CONSTRAINTS

Required fields:

```
-- User must have email and password
ALTER TABLE users
ALTER COLUMN email SET NOT NULL,
ALTER COLUMN hashed_password SET NOT NULL;
-- Message must have content and role
ALTER TABLE messages
ALTER COLUMN content SET NOT NULL,
ALTER COLUMN role SET NOT NULL;
-- Portfolio must have stocks
```

ALTER TABLE portfolios

ALTER COLUMN stocks SET NOT NULL;

3.3.5 DEFAULT VALUES

Sensible defaults for optional fields:

-- New users are beginners by default

ALTER TABLE users

ALTER COLUMN user_type SET DEFAULT 'beginner',

ALTER COLUMN risk_tolerance SET DEFAULT 'moderate';

-- Conversations start with 0 messages

ALTER TABLE conversations

ALTER COLUMN total_messages SET DEFAULT 0;

-- Messages default to text input

ALTER TABLE messages

ALTER COLUMN input_type SET DEFAULT 'text';

4. CORE COMPONENTS DESIGN

4.1 QUERY ROUTER AGENT

PURPOSE: Understand user intent and route to the appropriate specialist agent

RESPONSIBILITY: "Traffic Controller of the System"

The Query Router is the FIRST agent that processes every user query. Its job is to understand WHAT the user wants and WHO should handle it.

4.1.1 Design Overview

CLASSIFICATION TASK: Map user query → Intent category → Target agent

INPUT:

- User query (text)
- Conversation history (optional)
- User profile (experience level)

OUTPUT:

- Intent classification (educational/market_data/portfolio/general)
- Confidence score (0-1)
- Recommended agent(s)
- Reasoning for the decision

4.1.2 INTENT CATEGORIES

The router classifies queries into 6 PRIMARY INTENTS:

1. EDUCATIONAL

- Explaining financial concepts
- Defining terms (P/E ratio, mutual funds, etc.)
- Teaching investment principles

Examples: "What is a mutual fund?"

Routes to: Educational Agent

2. MARKET DATA

- Current stock prices
- Company fundamentals
- Market trends and analysis

Examples: "What is the price of Apple stock?"

Routes to: Market Data Agent

3. PORTFOLIO_BUILDING

- Investment recommendations
- Portfolio allocation
- Risk-based suggestions

Examples: "Build me a portfolio”

Routes to: Portfolio Builder Agent

4. PORTFOLIO_ANALYSIS

- Analyzing existing portfolios
- Performance evaluation
- Rebalancing suggestions

Examples: "Analyze my current investments”

Routes to: Portfolio Builder Agent (analysis mode)

5. COMPARISON

- Stock vs stock comparisons
- Sector comparisons
- Investment option comparisons

Examples: "Compare Apple vs Microsoft"

Routes to: Multiple agents (Market + Educational)

6. GENERAL_QUERY

- Greetings, small talk
- System help
- Unclear intent

4.1.3 DECISION LOGIC

A step by step routing process:

1) Step1: Keyword detection

- It looks out for specific intent in the query. Say for education : “what is”, “explain”, “how does”, “define”.

2) Step2: Context analysis.

- Checks conversation history for context. If previous query was about AAPL

3) Step3: LLM based classification

- Use of DSPy + Gemini for semantic understanding.
- Prompt: “Classify the intent of :{user query}, User level: {beginner}
- Output: Intent + Confidence+ Reasoning.

4) Step4: Confidence thresholding

- If confidence >= 0.7: Route to single agent
- If confidence < 0.7: Route to multiple agents
- If confidence < 0.4: Ask clarifying question

5) Step5: Agent selection

- Return: {agent_name, confidence, reasoning}

4.2 EDUCATIONAL CONTENT AGENT

PURPOSE: Explain financial concepts adaptively based on the user's expertise level

RESPONSIBILITY: "The Patient Teacher"

The Educational Agent transforms complex financial jargon into clear, Understandable explanations tailored to the user's level of knowledge.

4.2.1 DESIGN OVERVIEW

1. INPUT:

- Financial concept/term to explain
- User experience level (beginner/intermediate/advanced)
- Optional: Specific aspect user wants to understand

2. OUTPUT:

- Clear explanation at appropriate level
- Real-world examples
- Related concepts to explore
- Visual metaphors

4.2.2 KNOWLEDGE BASE

STORAGE: PostgreSQL educational content table with vector embeddings

RETRIEVAL STRATEGY:

1. EXACT MATCH SEARCH

- User asks: "What is P/E ratio?"
- SQL: SELECT * FROM educational_content

  WHERE LOWER(title) LIKE '%p/e ratio%'

- Fast, precise for known terms

2. SEMANTIC SEARCH (RAG)

- User asks: "How do I value a company?"
- Generate query embedding
- Find similar content: "P/E ratio", "DCF valuation", "Book value"
- Combine multiple relevant pieces

3. HIERARCHICAL LEARNING PATH

- Beginner asks: "Stock market basics"
- Provide learning sequence
    - What is a stock?
    - How stock markets work
    - How to buy stocks
    - Risk and diversification
    - Building a portfolio

4.2.3 TOOLS USED BY EDUCATIONAL AGENT

1. search_educational_content()

- Searches educational_content table
- Vector similarity search
- Returns top K relevant pieces

2. get_learning_path()

- Generates prerequisite chain
- Example: Options → Derivatives → Stocks → Markets

3. generate_example()

- Creates contextual examples
- Uses current market data for relevance
- Indian market focus

4.3 MARKET DATA AGENT

PURPOSE: Fetch, interpret, and present real-time market data and analysis

RESPONSIBILITY: "The Market Analyst"

The Market Data Agent connects to external APIs (Yahoo Finance) to provide accurate, up-to-date information about stocks, indices, and market conditions.

4.3.1 DESIGN OVERVIEW

DATA RETRIEVAL + ANALYSIS TASK: Stock Symbol → Comprehensive Market Analysis

1) INPUT:
    a) Stock symbol (e.g., "AAPL", "HDFCBANK")
    b) Analysis type (price/fundamentals/technical/news)

    c) Time range (day/week/month/year)
2) OUTPUT:
    a) Current price and change
    b) Fundamental metrics (P/E, market cap, etc.)
    c) Technical indicators (RSI, moving averages)
    d) Recent news and sentiment
    e) Interpretation and recommendation

4.3.2 DATA SOURCES

PRIMARY: Yahoo Finance API (via yfinance library)

ADVANTAGES:

- Free, no API key required
- Comprehensive data (price, fundamentals, news)
- Global coverage (US + Indian markets)
- Real-time (15-20 min delay for free tier)
- Historical data available

LIMITATIONS:

- 15-20 minute delay (not truly real-time)
- Rate limits (~2000 requests/hour)
- Occasional data quality issues
- Limited to public market data

4.3.3 DATA RETRIEVAL ARCHITECTURE

Layered caching strategy:

1) In-Memory cache
    a) Faster, but temporary.
    b) Stores recent queries.
    c) Reduces API calls for repeated queries.
2) Database cache
    a) Stores in message table.
    b) Includes response data field.
    c) Enables historical analysis.
3) Yahoo finance API
    a) Fresh data from source.
    b) Subject to rate limits.

4.4 PORTFOLIO BUILDER AGENT

PURPOSE: Generate diversified investment portfolios based on risk profile

RESPONSIBILITY: "The Investment Strategist"

The Portfolio Builder Agent creates personalized stock portfolios that match the user's risk tolerance, investment amount, and financial goals.

4.4.1 DESIGN OVERVIEW

PORTFOLIO GENERATION TASK: Risk Profile + Goals → Diversified Portfolio

INPUT:

- Risk tolerance (low/moderate/high)
- Investment amount (optional)
- Financial goals (retirement/wealth/income)
- Time horizon (short/medium/long term)
- Sector preferences (optional)

OUTPUT:

- List of stocks with allocation percentages
- Sector distribution
- Expected return and risk metrics
- Diversification score
- Rationale for each selection

4.4.2 STOCK UNIVERSE

CURATED STOCK LIST: Top Indian stocks across sectors

Our portfolio builder selects from a pre-vetted universe of quality stocks:

BANKING & FINANCIAL (8 stocks):

- HDFCBANK.NS - HDFC Bank
- ICICIBANK.NS - ICICI Bank
- SBIN.NS - State Bank of India
- AXISBANK.NS - Axis Bank
- KOTAKBANK.NS - Kotak Mahindra Bank
- BAJFINANCE.NS - Bajaj Finance
- BAJAJFINSV.NS - Bajaj Finserv
- HDFCLIFE.NS - HDFC Life Insurance

IT SERVICES (6 stocks):

- TCS.NS - Tata Consultancy Services
- INFY.NS - Infosys
- WIPRO.NS - Wipro
- HCLTECH.NS - HCL Technologies
- TECHM.NS - Tech Mahindra
- LTI.NS - L&T Infotech

FMCG (5 stocks):

- HINDUNILVR.NS - Hindustan Unilever
- ITC.NS - ITC Limited
- NESTLEIND.NS - Nestle India
- BRITANNIA.NS - Britannia Industries
- DABUR.NS - Dabur India

ENERGY & UTILITIES (4 stocks):

- RELIANCE.NS - Reliance Industries
- ONGC.NS - Oil and Natural Gas Corp
- NTPC.NS - NTPC Limited
- POWERGRID.NS - Power Grid Corporation

PHARMACEUTICALS (4 stocks):

- SUNPHARMA.NS - Sun Pharmaceutical
- DRREDDY.NS - Dr. Reddy's Laboratories
- CIPLA.NS - Cipla
- DIVISLAB.NS - Divi's Laboratories

AUTO & ANCILLARIES (4 stocks):

- MARUTI.NS - Maruti Suzuki
- TATAMOTORS.NS - Tata Motors
- HEROMOTOCO.NS - Hero MotoCorp
- BAJAJ-AUTO.NS - Bajaj Auto

TOTAL: 31 stocks across 6 major sectors

Selection criteria:

- Large-cap stocks
- High liquidity
- Consistent profitability

4.4.3 PORTFOLIO ALLOCATION STRATEGIES

Three allocation methods

Equal weights

- ❖ All stocks will get equal allocation.
- ❖ Simple, easy to understand
- ❖ Good for beginners.

4.4.4 PORTFOLIO GENERATION ALGORITHM

Step by step process:

1. Determine asset allocation by risk based on risk tolerance and decide sector weights.
2. Select stocks from each sector by fetching top stocks from each sector. By filtering by market cap and liquidity.
3. Calculating stock allocation based on equal weight within sector.
4. Calculate expected return and risk by calculating expected return= weighted average of historical returns.
5. Use of herfindahl hirschman index and calculate diversification score.
6. Generate rationale for each stock and how it fits the portfolio.
7. Store portfolio in database and eventually save to portfolio table.

4.5 MULTIAGENT ORCHESTRATOR

PURPOSE: Coordinate multiple agents for complex queries requiring multiple

perspectives

RESPONSIBILITY: "The Project Manager"

The Multi-Agent Orchestrator is the brain that decides WHICH agents to invoke,

WHEN to invoke them, and HOW to combine their responses into a coherent answer.

4.5.1 DESIGN OVERVIEW

COORDINATION TASK: Complex Query → Multiple Agents → Synthesized Response

The orchestrator handles queries that require expertise from multiple agents:

- "Should I invest in Apple stock?" → Market Data + Portfolio Builder
- "Explain P/E ratio and show me examples" → Educational + Market Data

- "Compare HDFC vs ICICI for my portfolio" → Market Data (×2) + Portfolio Builder

INPUT:

- User query
- Intent classification from Query Router
- User profile (risk tolerance, experience level)
- Conversation context

OUTPUT:

- Coordinated agent execution plan
- Combined responses from multiple agents
- Synthesized, coherent final answer
- Metadata (which agents contributed, execution time)

KEY RESPONSIBILITIES:

1. COMPLEXITY ASSESSMENT: Determine if query needs 1 agent or multiple agents
2. AGENT SELECTION: Choose which specialized agents to involve
3. EXECUTION ORCHESTRATION: Run agents in parallel or sequence as needed
4. RESULT SYNTHESIS: Combine multiple agent responses coherently
5. CONFLICT RESOLUTION: Handle contradictory recommendations

4.5.2 COMPLEXITY ASSESSMENT ALGORITHM

The orchestrator first assesses query complexity to determine approach.

Simple level.

- Clear single intent.
- One agent can fully answer.
- No need for cross referencing.
- Example: What is a mutul fund?, Show me apple stock price, build me a low risk portfolio.
- Action: Route to single agent.

Moderate level.

- Multiple related intents.
- Agents need each others output.
- Sequential dependency.
- Example : Explain diversification and build me a portfolio

- Action:Educational first, then portfolio builder.

Complex.

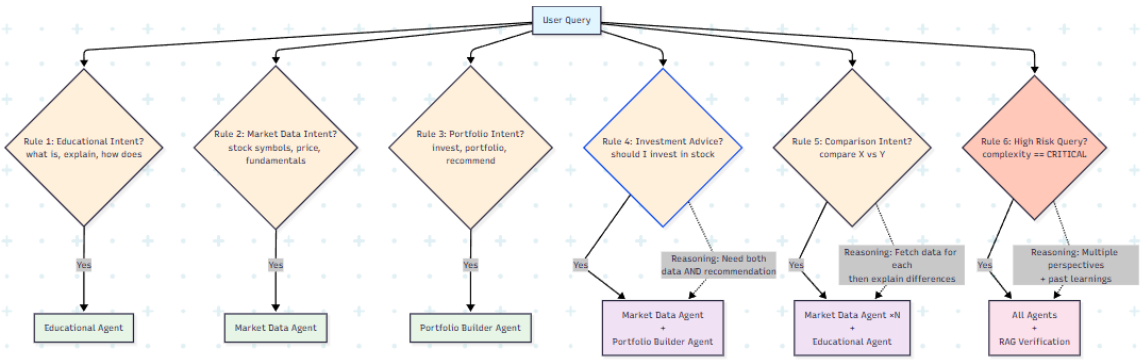
- Multiple independent intents.
- Agents can work simultaneously.
- Results need synthesis.
- Example: Should I invest in tech stocks?
- Action:Market data, Educational and portfolio builder.

Critical

- High stakes financial decision.
- Multiple perspectives needed.
- Verification and self reflection required.
- Example: Should I invest my entire savings in crypto?
- Action: All agents+ RAG verification+warnings.

4.5.3 AGENT SELECTION STRATEGY

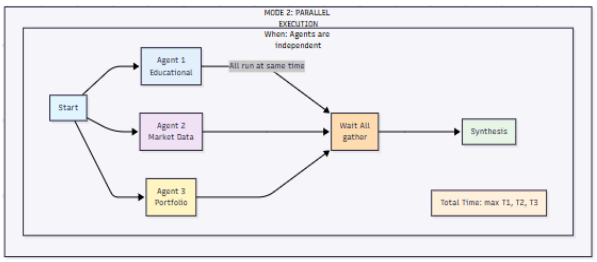
Based on complexity and intent, select appropriate agents.

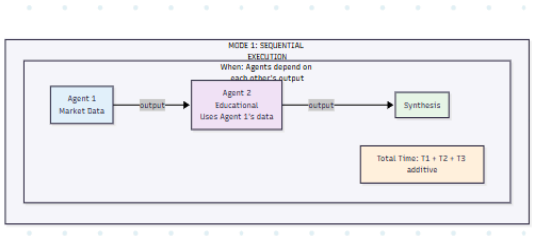


4.5.4 EXECUTION ORCHASTRATION

Once agents are selected, the orchestrator executes them efficiently.

Two execution models





4.5.5 RESULT SYNTHESIS

After agents complete, orchestrator synthesizes responses into coherent answer.

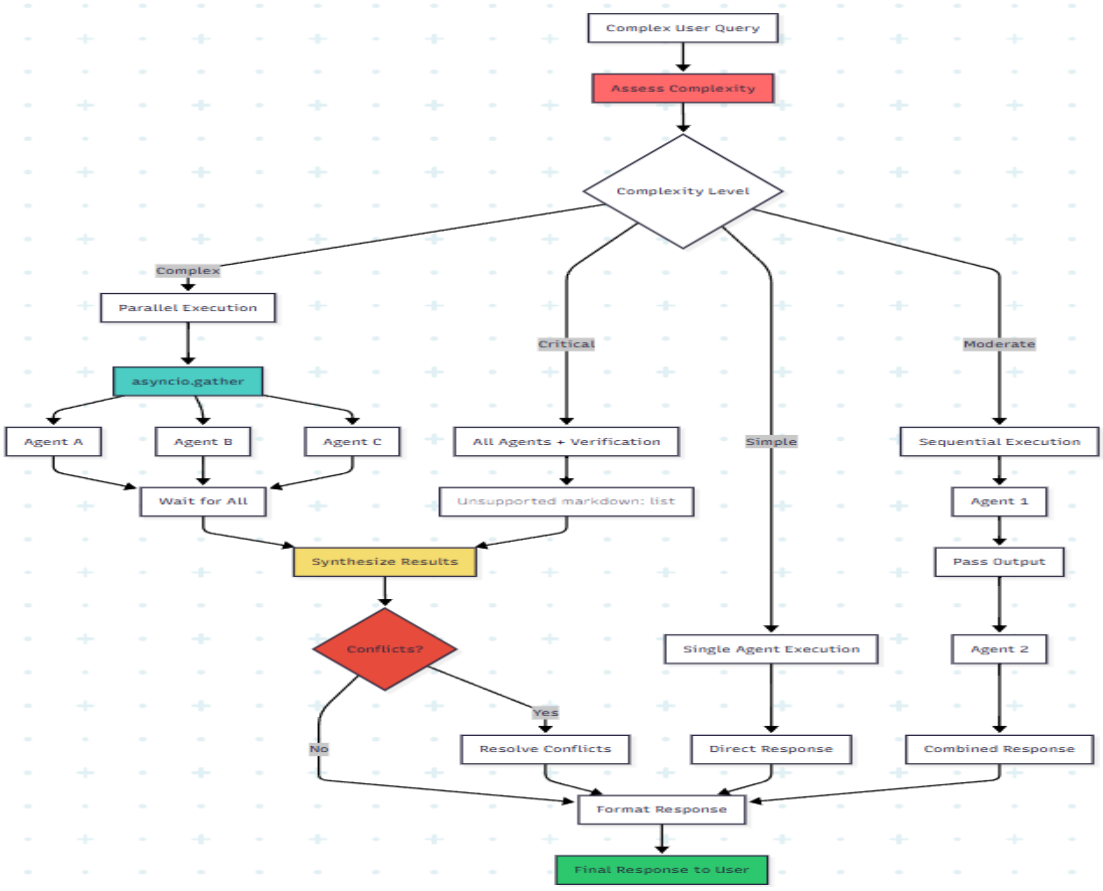
1. DIFFERENT FORMATS
    a. Educational: Long explanations
    b. Market Data: Numbers and charts
    c. Portfolio: Stock lists and allocations
    d. Solution: Use structured templates
2. POTENTIALLY CONTRADICTORY
    a. Market Data says "Stock is expensive (P/E = 45)"
    b. Portfolio Builder says "Include in growth portfolio"
    c. Solution: Acknowledge both perspectives
3. INFORMATION OVERLOAD
    a. 3 agents × 500 words each = 1500 words
    b. Too much for user!
    c. Solution: Summarize key points
4. COHERENT NARRATIVE
    a. Responses should flow naturally
    b. Not just concatenated outputs

4.5.6 CONFLICT RESOLUTION

Sometimes agents provide contradictory advice. The orchestrator must handle this.
CONFLICT SCENARIOS:

1. Scenario 1: VALUATION DISAGREEMENT
    a. Market Data: "Stock is overvalued (P/E = 50)"
    b. Portfolio Builder: "Include in high-growth portfolio
    c. Resolution:"While the stock appears expensive based on traditional valuation (P/E ratio), it may fit in a high-growth portfolio if you're comfortable with higher risk. This is a classic growth vs value trade-off".
2. Scenario 2: RISK ASSESSMENT DISAGREEMENT
    a. Educational Agent: "High-risk investment"
    b. Portfolio Builder: "Suitable for moderate risk"
    c. Resolution:"Risk depends on context. For a single stock, this is high-risk. But as 10% of a diversified portfolio, it represents moderate risk."
3. Scenario 3: TIMING DISAGREEMENT
    a. Market Data: "RSI = 75 (overbought), wait for correction"
    b. Portfolio Builder: "Good long-term hold regardless of timing"
    c. Resolution: Distinguish time horizons:"For short-term traders, waiting for a dip makes sense. For long-term".

4.5.7 MULTI_AGENT ORCHESTRATOR FLOW



5. DATA FLOW AND INTERACTIONS

5.1 USER QUERY PROCESSING FLOW

HIGH-LEVEL FLOW:

User Request → Authentication → Chat Router → Agentic RAG → Multi-Agent Orchestrator → Specialized Agents → Result Synthesis → Database Storage → Response to User

TOTAL LATENCY TARGET: < 5 seconds for most queries

STEP-BY-STEP FLOW: SIMPLE QUERY

EXAMPLE QUERY: "What is a mutual fund?"
Expected Route: Educational Agent (single agent, simple)

1. HTTP REQUEST ARRIVAL - Client → FastAPI Server
2. AUTHENTICATION MIDDLEWARE - Verify JWT Token

3. CHAT ROUTER - REQUEST VALIDATION
4. FETCH USER PROFILE - Get user preferences and experience level
5. STORE USER MESSAGE IN DATABASE - Save the incoming message
6. AGENTIC RAG - INTENT CLASSIFICATION
7. AGENTIC RAG - RETRIEVAL PLANNING - Based on intent
8. AGENTIC RAG - VECTOR SIMILARITY SEARCH - Retrieve relevant context from database
9. ORCHESTRATOR - COMPLEXITY ASSESSMENT
10. ORCHESTRATOR - AGENT SELECTION
11. EDUCATIONAL AGENT EXECUTION
12. AGENT OUTPUT FORMATTING - Format the agent response
13. STORE AI RESPONSE IN DATABASE - Save assistant message with metadata
14. HTTP RESPONSE TO CLIENT - Return JSON response to user

STEP-BY-STEP FLOW: COMPLEX QUERY

EXAMPLE QUERY: "Should I invest in Apple stock?"

Expected Route: Market Data Agent + Portfolio Builder (multi-agent, parallel)

1. STEPS 1-7: Same as Simple Query (Authentication, Validation, RAG Intent, Retrieval Planning)
2. STEP 8: INTENT CLASSIFICATION
    a. Query: "Should I invest in Apple stock?"
    b. LLM Classification:
        i. Primary Intent: PERSONALIZED_ADVICE
        ii. Secondary Intents: MARKET_DATA (need Apple stock info), PORTFOLIO_BUILDING (investment recommendation)
3. STEP 9: COMPLEXITY ASSESSMENT - Higher complexity than simple query
    a. Multiple intents: market_data + portfolio
4. STEP 10: AGENT SELECTION - Multiple agents needed
5. STEP 11: PARALLEL AGENT EXECUTION - Three agents execute at the same time using asyncio.gather()
6. STEP 12: RESULT SYNTHESIS - Combine three agent outputs into coherent response
7. STEP 13-14: Store Response & Return to User - Same as simple query

5.2 MULTI-AGENT EXECUTION FLOW

5.2.1 ORCHESTRATION MODES

The orchestrator supports THREE execution modes based on agent dependencies:

1. DIRECT (Single Agent)
    a. When: Simple query, single intent
    b. Flow: Query → Agent → Response
    c. Latency: Minimal overhead
    d. Example: "What is a stock?" → Educational Agent only
2. SEQUENTIAL (Dependent Agents)
    a. When: Agent B needs output from Agent A
    b. Flow: Query → Agent A → Output A → Agent B → Response
    c. Latency: Additive (A + B)
    d. Example: "Explain P/E ratio and show me Apple's" → Market Data Agent (get Apple P/E) → Educational Agent (explain using actual number)
3. PARALLEL (Independent Agents)
    a. When: Agents can work independently
    b. Flow: Query → [Agent A || Agent B || Agent C] → Synthesize → Response
    c. Latency: max(A, B, C) - fastest mode!
    d. Example: "Should I invest in tech stocks?"→ Market Agent || Portfolio Agent || Educational

5.2.2 DEPENDENCY DETECTION

DECISION TREE:

- Check 1: Do agents need same input?
    - If YES → Can run in PARALLEL
    - If NO → Need to check dependencies
- Check 2: Does Agent B reference Agent A's output?
    - If YES → Must run SEQUENTIAL (A then B)
    - If NO → Can run in PARALLEL
- Check 3: Any explicit ordering requirement?
    - If YES → Run in specified order (SEQUENTIAL)
    - If NO → Default to PARALLEL for speed

EXAMPLES:

Query: "Compare Apple vs Microsoft"
Analysis:

- Agent A: Market Data (AAPL)
- Agent B: Market Data (MSFT)
- Dependencies: NONE (both fetch different stocks)
- Decision: PARALLEL

Query: "Fetch Tesla price and explain if it's good"

Analysis:

- Agent A: Market Data (TSLA)
- Agent B: Educational (explain P/E ratio)
- Dependencies: B needs A's P/E number to explain
- Decision: SEQUENTIAL (A → B)

Query: "Should I invest $10,000 in SBI Bank?"
Analysis:

- Agent A: Market Data (SBIN.NS)
- Agent B: Portfolio Builder (recommendation)
- Agent C: Educational (explain diversification)
- Dependencies: NONE (all use same query)
- Decision: PARALLEL

5.2.3 RESULT AGGREGATION PATTERNS

PATTERN 1: CONCATENATION (Simple)

Used when: Agents provide independent insights
Method: Stack responses in logical order
Example: Market Data output: "Apple trading at $150..." + Educational output: "Diversification means..." + Portfolio output: "Recommendation: 10-15% allocation" = Combined: Three separate sections

PATTERN 2: SYNTHESIS (Advanced)

Used when: Agents provide overlapping insights
Method: LLM combines into coherent narrative
Example: Market Data: "P/E is 28.5", Educational: "P/E ratio measures valuation" → Synthesized: "Apple's P/E of 28.5 means investors pay $28.50 for each $1 of earnings, indicating moderate expectations..."

PATTERN 3: PRIORITIZATION (Conflict)

Used when: Agents contradict each other
Method: Present both views + explain nuance
Example: Market Data: "Stock overvalued (P/E high)", Portfolio: "Good for growth portfolio" → Resolved: "While expensive by traditional metrics, fits growth strategy if you accept risk"

5.3 RAG RETRIEVAL FLOW

5.3.1 RAG SYSTEM ARCHITECTURE

The RAG system operates in 5 DISTINCT PHASES:

- PHASE 1: INTENT CLASSIFICATION - Understand WHAT user wants to know
- PHASE 2: RETRIEVAL PLANNING - Decide WHERE to search and HOW
- PHASE 3: EMBEDDING GENERATION - Convert query to vector representation
- PHASE 4: VECTOR SIMILARITY SEARCH - Find semantically similar content
- PHASE 5: RELEVANCE VERIFICATION - Filter and rank retrieved documents

5.3.2 RAG PERFORMANCE OPTIMIZATION TECHNIQUES

- TECHNIQUE 1: EMBEDDING CACHING
    - Problem: Generating embeddings is slow
    - Solution: Cache common queries
- TECHNIQUE 2: APPROXIMATE NEAREST NEIGHBOR (ANN)
    - Problem: Exact vector search slow on large datasets
    - Solution: Use ANN algorithms (ivfflat, HNSW)
- TECHNIQUE 3: SMART FILTERING
    - Problem: Searching entire DB is wasteful
    - Solution: Pre-filter before vector search
- TECHNIQUE 4: PARALLEL RETRIEVAL
    - Problem: Sequential searches are slow
    - Solution: Fetch from multiple sources in parallel
- TECHNIQUE 5: RESULT TRUNCATION
    - Problem: Large context wastes tokens
    - Solution: Retrieve top K, send only relevant

5.4 DATABASE INTERACTION FLOW

5.4.1 DATABASE CONNECTION ARCHITECTURE

CONNECTION POOLING STRATEGY:

- Technology: SQLAlchemy async + asyncpg
- Pool Size: 20 connections
- Overflow: 10 additional connections
- Timeout: 30 seconds
- Recycle: 3600 seconds (1 hour)

Need for Connection Pooling:

- Without Pooling:
    - Each request opens new connection → Slow
    - Thousands of requests → Thousands of connections → DB overload
- With Pooling:
    - Reuse existing connections → Fast (~1ms)
    - Max 30 concurrent connections → Controlled load
    - Automatic cleanup → No connection leaks

5.4.2 COMMON DATABASE OPERATIONS

- OPERATION 1: CREATE USER MESSAGE
    - Action: Store incoming user message with embedding
    - Tables: messages
    - Transaction: Single INSERT
- OPERATION 2: STORE AI RESPONSE WITH METADAT
    - Action: Store AI response with agent metadata
    - Tables: messages, conversations (update)
    - Transaction: INSERT + UPDATE
- OPERATION 3: VECTOR SIMILARITY SEARCH
    - Action: Find similar messages using vector search
    - Tables: messages
    - Transaction: SELECT with vector operator
- OPERATION 4: CREATE PORTFOLIO (TRANSACTION)
    - Action: Generate and store portfolio
    - Tables: portfolios, messages (link)
    - Transaction: Multi-step with rollback capability

6. API DESIGN

6.1 API DESIGN PRINCIPLES

1. RESTful Architecture
    a. Resource-based URLs
    b. Standard HTTP methods (GET, POST, PUT, DELETE)
    c. Stateless communication
    d. JSON request/response format
2. Authentication
    a. JWT (JSON Web Tokens) for session management
    b. Bearer token in Authorization header
    c. Token expiration: 24 hours
    d. Refresh token mechanism
3. Versioning

    a. URL-based versioning: /api/v1/…
    b. Backward compatibility maintained
    c. Deprecated endpoints clearly marked
4. Rate Limiting
    a. 100 requests per minute per user
    b. 1000 requests per hour per user
    c. Burst allowance: 10 requests/second
5. Error Handling
    a. Consistent error response format
    b. HTTP status codes follow RFC standards
    c. Detailed error messages for debugging

6.2 TECHNOLOGY STACK

1. Framework: FastAPI 0.104.1
    a. Async/await support for high concurrency
    b. Automatic OpenAPI documentation
    c. Built-in request validation
    d. Type hints with Pydantic
2. Authentication: JWT (PyJWT)
    a. HS256 algorithm
    b. Secret key rotation support
    c. Token blacklisting for logout
3. Validation: Pydantic v2
    a. Automatic request validation
    b. Type safety
    c. Custom validator

7. SECURITY AND PRIVACY DESIGN

7.1 AUTHENTICATION AND AUTHORIZATION

7.1.1 JWT-BASED AUTHENTICATION

APPROACH: Stateless token-based authentication

SECURITY FEATURES:

- HS256 algorithm (HMAC with SHA-256)
- 256-bit secret key (stored in environment variables)
- Token expiration: 24 hours (access), 30 days (refresh)
- Token ID (jti) for revocation tracking
- Refresh token rotation on use

7.1.2 PASSWORD SECURITY

HASHING: bcrypt with cost factor 12

Need for Bcrypt

- Designed for password hashing
- Adaptive cost factor (can increase over time)
- Includes salt automatically
- Resistant to rainbow table attacks
- Slow by design (prevents brute force)

PASSWORD POLICY:

- Minimum 8 characters
- At least 1 uppercase letter
- At least 1 lowercase letter
- At least 1 digit
- At least 1 special character (optional but recommended)

7.1.2 ROLE-BASED ACCESS CONTROL (RBAC)

ROLES:

- USER: Regular user (default)
- ADMIN: Administrative access

PERMISSIONS:

USER role:

- Create/read own conversations
- Generate portfolios
- Access chat API
- Update own profile

ADMIN role:

- All USER permissions
- View all users
- Manage educational content
- View system metrics
- Manage user accounts

7.2 DATA PROTECTION STRATEGY

7.2.1 DATA ENCRYPTION

ENCRYPTION AT REST:

- Database: PostgreSQL with transparent data encryption (TDE)
- Sensitive fields: Additional application-level encryption
- API keys: Encrypted in database, decrypted at runtime

ENCRYPTION IN TRANSIT:

- HTTPS/TLS 1.3 for all API communication
- Database connections over SSL
- Certificate validation enforced

7.2.2 DATA PRIVACY

PERSONAL DATA HANDLING:

- Minimal data collection (only essential fields)
- User consent required for data processing
- Clear privacy policy
- Data retention policies

DATA COLLECTED:

- Email (authentication)
- Name (personalization)
- Phone (optional, for notifications)
- Date of birth (optional, for age verification)
- Financial preferences (risk tolerance, etc.)

SENSITIVE DATA PROTECTION:

- Portfolio data: Owned by user, isolated per user
- Conversation history: Private, not shared
- Financial data: Not stored (fetched in real-time)

DATA ACCESS CONTROLS:

- Users can only access their own data
- Queries filtered by user_id
- Row-level security in database

7.2.3 DATA RETENTION AND DELETION

RETENTION POLICIES:

- Active user data: Retained indefinitely
- Inactive users (>2 years): Anonymized or deleted
- Conversation history: Retained for 1 year
- System logs: Retained for 90 days

RIGHT TO DELETION (GDPR):

- Users can request account deletion

7.3 API SECURITY

7.3.1 RATE LIMITING

PURPOSE: Prevent abuse and DoS attacks

LIMITS:

- Authentication endpoints: 10 requests/minute per IP
- Chat API: 100 requests/minute per user
- General API: 1000 requests/hour per user

7.3.2 INPUT VALIDATION & SANITIZATION

PYDANTIC MODELS: Automatic validation

SQL INJECTION PREVENTION:

- Use SQLAlchemy ORM (parameterized queries)
- Never concatenate user input into SQL
- Validate all inputs before database operations

8. TECHNOLOGICAL JUSTIFICATION

8.1 BACKEND FRAMEWORK: FASTAPI

FastAPI chosen for performance + productivity

1. PERFORMANCE
    a. Built on Starlette and Pydantic
    b. Async/await support for high concurrency
    c. One of the fastest Python frameworks
    d. Comparable to Node.js and Go in benchmarks
2. DEVELOPER PRODUCTIVITY
    a. Automatic API documentation (Swagger/OpenAPI)

    b. Type hints with runtime validation
    c. Minimal boilerplate code
    d. Excellent error messages
3. MODERN PYTHON FEATURES
    a. Python 3.7+ type hints
    b. Pydantic for data validation
    c. Async support throughout
4. ECOSYSTEM
    a. Large community
    b. Excellent documentation
    c. Wide adoption in industry

ALTERNATIVES CONSIDERED:

- Flask: Mature but lacks async, requires more boilerplate
- Django: Too heavyweight for API-only service
- Express.js (Node): Would require learning new language

8.2 LLM FRAMEWORK: DSPY

DSPy for structured, maintainable LLM interactions

1. STRUCTURED PROMPTING
    a. Signatures define input/output clearly
    b. Type-safe interactions with LLM
    c. Easier to maintain than string templates
2. MODULARITY
    a. Compose complex pipelines from modules
    b. Reusable components (ChainOfThought, ReAct, etc.)
    c. Easy to test individual components
3. OPTIMIZATION
    a. Can optimize prompts automatically
    b. Few-shot learning support
    c. Evaluation-driven improvement
4. ABSTRACTION
    a. LLM-agnostic (can switch from Gemini to GPT)
    b. Hides API-specific details
    c. Consistent interface

ALTERNATIVES CONSIDERED:

- LangChain: More complex, heavier, harder to optimize
- Plain LLM API: Too low-level, no structure

- Guidance: Less mature than DSPy

8.3 LLM PROVIDER: GOOGLE GEMINI

Gemini for cost-effectiveness and ease of integration

1. COST-EFFECTIVENESS
    a. Free tier: 1500 requests/day
    b. Competitive pricing for paid tier
    c. Good performance-to-cost ratio
2. PERFORMANCE
    a. Fast response times (gemini-flash optimized)
    b. Good quality for most queries
    c. Multimodal capabilities (future: image analysis)
3. INTEGRATION
    a. Official Python SDK
    b. Embedding model included (embedding-001)
    c. Same provider for text and embeddings
4. CONTEXT WINDOW
    a. 32K tokens input (sufficient for most queries)
    b. Can handle long conversation histories

ALTERNATIVES CONSIDERED:

- OpenAI GPT-4: More expensive, better quality (overkill for MVP)
- Claude: Good but pricing less competitive
- Llama 2: Self-hosting complexity not worth it for MVP
- Mistral: Smaller community, less mature

8.4 DATABASE: POSTGRESQL + PGVECTOR

PostgreSQL + pgvector for simplicity and performance

1. POSTGRESQL
    a. Robust, mature, production-proven
    b. ACID compliance for data integrity
    c. Rich data types (JSONB, Arrays, etc.)
    d. Excellent performance for complex queries
    e. Strong community and ecosystem
2. PGVECTOR EXTENSION
    a. Native vector similarity search
    b. Eliminates need for separate vector database
    c. Reduces infrastructure complexity

    d. Atomic operations (vector + relational in one transaction)
    e. Cost-effective (no additional service)
3. UNIFIED DATA STORE
    a. User data, conversations, and vectors in one DB
    b. Simplified backup/restore
    c. Lower latency (no network hop to vector DB)
    d. Easier to maintain

ALTERNATIVES CONSIDERED:

- MySQL: Less advanced features than PostgreSQL
- MongoDB: NoSQL not ideal for relational data
- Pinecone/Weaviate: Separate vector DB adds complexity
- Chroma: Embedded DB, not production-ready for scale

8.5 MARKET DATA: YAHOO FINANCE

Yahoo Finance for free, comprehensive data

1. FREE & ACCESSIBLE
    a. No API key required
    b. No cost for reasonable usage
    c. Ideal for MVP and education
2. COMPREHENSIVE DATA
    a. Real-time quotes (15-min delay)
    b. Historical data
    c. Fundamentals (P/E, dividends, etc.)
    d. Analyst recommendations
3. EASE OF USE
    a. Simple Python library
    b. Well-documented
    c. Active maintenance
4. SUFFICIENT FOR USE CASE
    a. Educational/advisory platform (not trading)
    b. 15-minute delay acceptable
    c. Focus on Indian market covered

LIMITATIONS & MITIGATION:

- Unofficial API (could change): Cache data, have fallback
- Rate limits: Implement caching, respectful usage
- Best-effort availability: Error handling, graceful degradation

ALTERNATIVES CONSIDERED:

- Alpha Vantage: Free tier too limited (25 req/day)
- NSE/BSE APIs: Complex authentication, limited data

8.6 ASYNC PROCESSING: PYTHON ASYNCIO

Asyncio for efficient concurrent I/O operations

1. CONCURRENCY WITHOUT THREADS
    a. Single-threaded async model
    b. Efficient for I/O-bound tasks
    c. No race conditions, simpler debugging
2. NATIVE PYTHON SUPPORT
    a. No external dependencies
    b. Works seamlessly with FastAPI
    c. Modern Python standard (3.7+)
3. PERFORMANCE
    a. Can handle thousands of concurrent connections
    b. Low memory overhead per task
    c. Efficient for API calls, DB queries
4. ECOSYSTEM
    a. asyncpg (async PostgreSQL driver)
    b. aiohttp (async HTTP client)
    c. Many libraries support async

8.7 ORM: SQLALCHEMY (ASYNC)

SQLAlchemy for maturity and async support

1. INDUSTRY STANDARD
    a. Most popular Python ORM
    b. Battle-tested in production
    c. Extensive documentation
2. ASYNC SUPPORT
    a. Native async/await support (v2.0+)
    b. Works with asyncpg driver
    c. Non-blocking database operations
3. FLEXIBILITY
    a. Can use ORM or raw SQL when needed
    b. Complex query support
    c. Relationship management
4. TYPE SAFETY

    a. Modern type hints support
    b. IDE autocomplete
    c. Fewer runtime errors
5. MIGRATIONS
    a. Alembic integration for schema migrations
    b. Version control for database schema

ALTERNATIVES CONSIDERED:

- Django ORM: Tied to the Django framework
- Tortoise ORM: Less mature, smaller community

8.8 TECHNOLOGY STACK SUMMARY

COMPONENT	TECHNOLOGY	PRIMARY REASON
Backend Framework	FastAPI	Performance + Async
Database	PostgreSQL	Robust + ACID
Vector Search	pgvector	Unified Data Store
LLM Framework	DSPy	Structured Prompts
LLM Provider	Google Gemini	Cost Effective
Embeddings	Gemini Embeddings	Same Provider
Market Data	Yahoo Finance	Free + Comprehensive
ORM	SQLAlchemy (async)	Industry Standard
Password Hashing	bcrypt	Designed for pwds
Authentication	JWT (PyJWT)	Stateless tokens
Input Validation	Pydantic	Type Safety
Async Runtime	Python Asyncio	Native Support
HTTP Client	httpx/aiohttp	Async HTTP
Migrations	Alembic	Schema Versioning

REFERENCES

[1] Choi, C., Kwon, J., Ha, J., Choi, H., Kim, C., Lee, Y., Sohn, J., & Lopez-Lira, A. (2025). FinDER: Financial dataset for question answering and evaluating retrieval-augmented generation. *arXiv preprint arXiv:2504.15800*. https://doi.org/10.48550/arXiv.2504.15800

[2] Xiao, Y., Sun, E., Luo, D., & Wang, W. (2024). TradingAgents: Multi-agents LLM financial trading framework. *arXiv preprint arXiv:2412.20138*. https://doi.org/10.48550/arXiv.2412.20138

[3] Lopez-Lira, A., Scheer, M., & Schwenkler, G. (2024) . Using Large Language Models for Financial Advice. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4850039

[4] Cao, S., Jiang, W., & Wang, J. (2024) - "Generative AI and Investor Processing of Financial Information. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5053905

[5] Angel One. (2025). *Online trading & stock broking in India*. https://www.angelone.in

[6] National Stock Exchange of India. (2025). *NSE - National Stock Exchange of India Ltd*. https://www.nseindia.com