Technical Architecture Document

Multi-Modal Financial Advisor Chatbot

Table of Contents

1. System Architecture

1.1 Architectural Overview

The Multi-Modal Financial Advisor Chatbot implements a layered microservices architecture designed for scalability, modularity, and resilience. The system is structured in distinct layers, each with specific responsibilities and well-defined interfaces that facilitate independent development and deployment.

```
+-----------------------------------------------------------------------------
|                                  CLIENT LAYER
| +---------------------------------------------------------------------------
| |                               WEB INTERFACE
| |                                (React.js)
| +---------------------------------------------------------------------------
+-----------------------------------------------------^-----------------------
                                                      |
                                                      | HTTP/REST
                                                      |
+-----------------------------------------------------v-----------------------
|                              APPLICATION LAYER
| +---------------------------------------------------------------------------
| |                             FASTAPI BACKEND
| |
| |  +---------------------+  +---------------------+  +-----------------------+
| |  |  ONBOARDING ENGINE |  |  AUTH SYSTEM        |  |  API ENDPOINTS        |
| |  |                    |  |                     |  |                       |
| |  | - Conversation     |  | - JWT-based auth    |  | - Chat endpoints      |
| |  |   flow management  |  | - Token generation |  | - Recommendation APIs |
| |  | - Profile creation |  | - Validation        |  | - User management     |
| |  |                    |  |                     |  |                       |
| |  +---------------------+  +---------------------+  +-----------------------+
| |              |                    |                        |
| |              +--------------------+------------------------+
| |                                   |
| +---------------------------------- | ----------------------------------------
+------------------------------------|----------------------------------------
                                      |
                                      | Internal APIs
                                      |
+-----------------------------------------------------v-----------------------
|                                  AI LAYER
|  +--------------------------+                        +-------------------------
|  |    ONBOARDING AGENT      |<---------------------->|     ADVISORY AGENT
|  |                          |                        |
|  | - Contextual Q&A         |                        | - Personalized recs
|  | - Risk assessment        |    context transfer    | - Educational content
|  | - Goal identification    |                        | - Product-specific info
|  | - Financial profiling    |                        | - Financial planning
|  +--------------------------+                        +-------------------------
|         ^           ^                                     ^             ^
|         |           |                                     |             |
|         v           v                                     v             v
|  +--------------------------+                        +-------------------------
|  |  MULTI-MODAL PROCESSOR   |                        |   RECOMMENDATION ENGINE
|  |                          |                        |
|  | - Text processing        |                        | - RAG system
|  | - Document analysis      |                        | - Meta-prompt generatic
|  | - Image recognition      |                        | - Matching algorithms
|  | - Future: voice          |                        | - RLHF
|  +--------------------------+                        +-------------------------
|
+-----------------------------------------------------------------------------
```

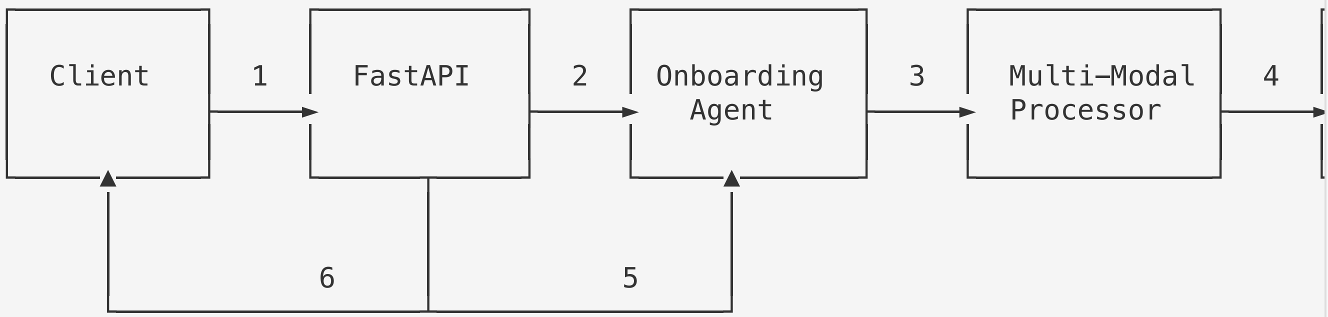
```
+------------------------------------------------------------------------------
|                                      DATA LAYER
|  +-------------------------+    +-------------------------+    +-------------
|  |        MONGODB          |    |       VECTOR STORE      |    |      REDIS C
|  |                         |    |                         |    |
|  | - User profiles         |    | - Embeddings           |    | - Session dat
|  | - Conversations        |    | - Semantic search      |    | - API response
|  | - Product catalog       |    | - Financial knowledge  |    | - Rate limiti
|  | - Transaction history   |    |   base vectors         |    | - Temporary s
|  +-------------------------+    +-------------------------+    +-------------
+------------------------------------------------------------------------------

+------------------------------------------------------------------------------
|                                    ETHICAL AI LAYER
|  +------------------------+    +-------------------------+    +-------------
|  |  EXPLAINABILITY TOOLS  |    |     BIAS DETECTION      |    |  TRANSPARENCY
|  |                        |    |                         |    |
|  | - Recommendation       |    | - Bias monitoring       |    | - User data a
|  |   justification        |    | - Fairness metrics      |    | - Model card
|  | - Decision path tracing|    | - Intervention systems  |    | - Audit loggi
|  +------------------------+    +-------------------------+    +-------------
+------------------------------------------------------------------------------
```

The architecture follows microservices principles, with each component having a single responsibility and communicating through well-defined interfaces. Key architectural decisions include:

- **Layered Separation:** Clear separation between layers allows for independent scaling and deployment of components.
- **Stateless Application Layer:** The FastAPI backend is designed to be stateless, facilitating horizontal scaling.
- **Dual-Agent Architecture:** Separation of concerns between onboarding and advisory functions improves specialization and maintainability.
- **Polyglot Persistence:** Different data storage technologies for different data types (MongoDB for documents, Vector Store for embeddings, Redis for caching).
- **Cross-Cutting Ethical AI Layer:** Ensures explainability, fairness, and transparency across all AI components.

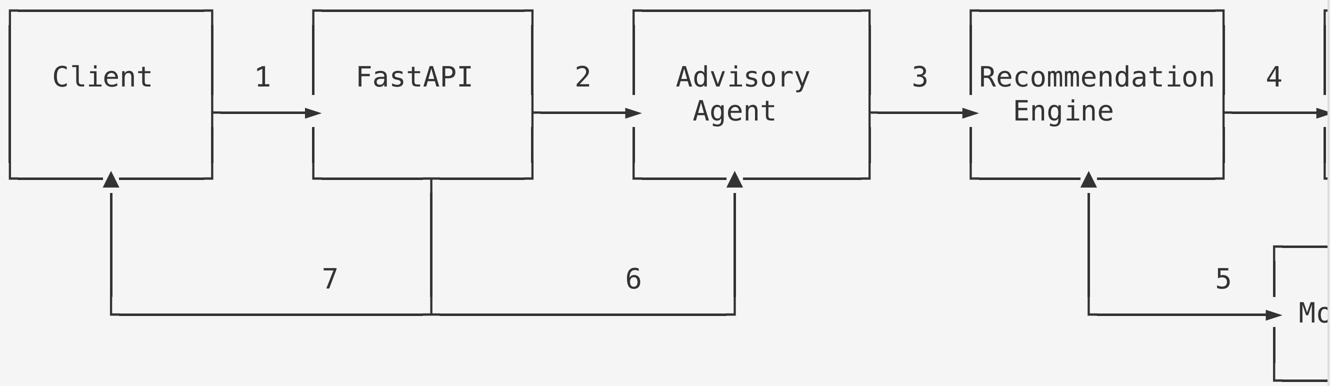
1.2 Detailed Component Interaction

1.2.1 Onboarding Flow



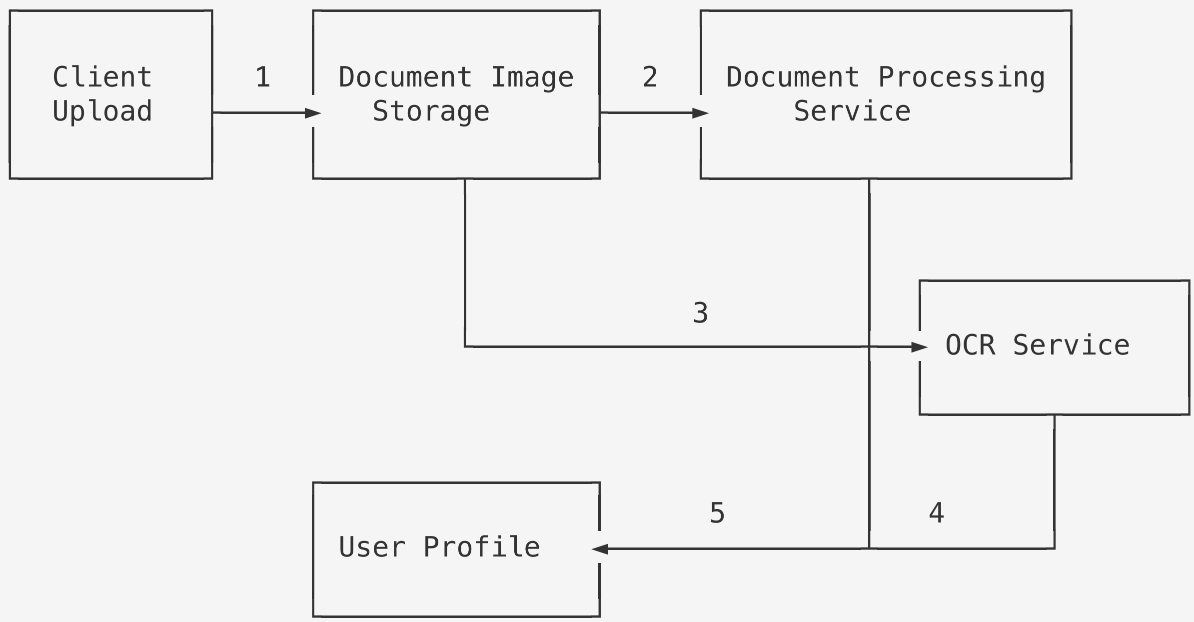

```
1. User sends initial data (text input or file upload)
2. FastAPI routes request to Onboarding Agent
3. For document/image analysis, Onboarding Agent calls Multi-Modal Processor
4. User profile data is stored in MongoDB
5. Onboarding Agent generates personalized follow-up questions
6. Response is returned to client
```

1.2.2 Advisory Flow



1. User sends query or request for recommendations
2. FastAPI routes to Advisory Agent
3. Advisory Agent requests personalized recommendations
4. Recommendation Engine retrieves relevant financial data from Vector Store
5. User profile and previous interactions are fetched from MongoDB
6. Advisory Agent generates personalized response with recommendations
7. Response returned to client with explanations

1.2.3 Multi-Modal Processing




```
1. User uploads financial document or image
2. File is temporarily stored
3. Document Processing Service dispatches to appropriate analyzer
4. Text is extracted and structured
5. Relevant financial data is extracted and added to user profile
```

These interaction diagrams illustrate the primary data flows within the system. Key aspects include:

- **Asynchronous Processing:** Long-running tasks like document analysis are processed asynchronously.
- **Context Preservation:** User context is maintained across interactions by storing and retrieving conversation history.
- **Progressive Profiling:** User profiles are continuously enriched based on interactions and provided documents.
- **LLM Selection:** Different LLMs are selected based on the specific task requirements and availability.

1.3 Design Patterns

The application incorporates multiple established design patterns to address common architectural challenges:

1.3.1 Repository Pattern

Used for database interactions, abstracting data access logic from business logic:

```
class UserRepository:
    def __init__(self, db_session):
        self.db_session = db_session

    async def get_by_id(self, user_id: str) -> Optional[User]:
        return await self.db_session.find_one({"_id": ObjectId(user_id)})

    async def create(self, user_data: dict) -> User:
        result = await self.db_session.insert_one(user_data)
        return await self.get_by_id(result.inserted_id)

    async def update(self, user_id: str, update_data: dict) -> Optional[User]:
        await self.db_session.update_one(
            {"_id": ObjectId(user_id)},
            {"$set": update_data}
        )
        return await self.get_by_id(user_id)
```

1.3.2 Factory Pattern

Used for LLM selection and creation based on task requirements and availability:

```
class LLMFactory:
    @staticmethod
    def create_llm(llm_type: str, **kwargs) -> BaseLLM:
        if llm_type == "openai":
            return OpenAILLM(api_key=settings.OPENAI_API_KEY, **kwargs)
        elif llm_type == "mistral":
            return MistralLLM(api_key=settings.MISTRAL_API_KEY, **kwargs)
        elif llm_type == "huggingface":
            return HuggingFaceLLM(token=settings.HUGGINGFACE_TOKEN, **kwargs)
        else:
            raise ValueError(f"Unsupported LLM type: {llm_type}")
```

1.3.3 Strategy Pattern

Used for implementing different recommendation strategies:

```
class RecommendationStrategy(ABC):
    @abstractmethod
    async def generate_recommendations(self, user_profile: dict) -> List[Recommendation]:
        pass

class ConservativeStrategy(RecommendationStrategy):
    async def generate_recommendations(self, user_profile: dict) -> List[Recommendation]:
        # Implementation for conservative investors

class AggressiveStrategy(RecommendationStrategy):
    async def generate_recommendations(self, user_profile: dict) -> List[Recommendation]:
        # Implementation for aggressive investors

class RecommendationEngine:
    def __init__(self):
        self.strategies = {
            "conservative": ConservativeStrategy(),
            "moderate": ModerateStrategy(),
            "aggressive": AggressiveStrategy()
        }

    async def get_recommendations(self, user_profile: dict) -> List[Recommendation]:
        risk_profile = user_profile.get("risk_tolerance", "moderate")
        strategy = self.strategies.get(risk_profile, self.strategies["moderate"])
        return await strategy.generate_recommendations(user_profile)
```

1.3.4 Observer Pattern

Used for events like user profile updates that may trigger multiple actions:

```
class ProfileUpdateEvent:
    def __init__(self):
        self._observers = []

    def register_observer(self, observer):
        self._observers.append(observer)

    def notify_observers(self, user_id: str, updated_fields: dict):
        for observer in self._observers:
            observer.update(user_id, updated_fields)

class RecommendationRefresher:
    def update(self, user_id: str, updated_fields: dict):
        # Queue a task to refresh recommendations based on profile changes

class UserActivityLogger:
    def update(self, user_id: str, updated_fields: dict):
        # Log the profile update activity

# Usage
profile_update_event = ProfileUpdateEvent()
profile_update_event.register_observer(RecommendationRefresher())
profile_update_event.register_observer(UserActivityLogger())

# When profile is updated
profile_update_event.notify_observers(user_id, updated_fields)
```

1.3.5 Decorator Pattern

Used for adding functionality to API endpoints, such as caching or rate limiting:

```
def cache_response(expiration_seconds: int = 300):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            # Generate a cache key based on function arguments
            cache_key = f"{func.__name__}:{hash(str(args))}{hash(str(kwargs))}"

            # Check if result is in cache
            cached_result = await redis_client.get(cache_key)
            if cached_result:
                return json.loads(cached_result)

            # Execute function and cache result
            result = await func(*args, **kwargs)
            await redis_client.set(
                cache_key,
                json.dumps(result),
                expiration_seconds
            )
            return result
        return wrapper
    return decorator

# Usage on an API endpoint
@router.get("/products/popular")
@cache_response(expiration_seconds=3600)  # Cache for 1 hour
async def get_popular_products():
    # Implementation that might be expensive to compute
    return await product_service.get_popular_products()
```

1.3.6 Middleware Pattern

Used for cross-cutting concerns like authentication, logging, and error handling:

```python
@app.middleware("http")
async def auth_middleware(request: Request, call_next):
    # Skip auth for certain endpoints
    if request.url.path in ["/api/auth/token", "/api/auth/register", "/docs"]:
        return await call_next(request)

    # Check for authorization header
    auth_header = request.headers.get("Authorization")
    if not auth_header or not auth_header.startswith("Bearer "):
        return JSONResponse(
            status_code=401,
            content={"detail": "Invalid authentication credentials"}
        )

    token = auth_header.replace("Bearer ", "")
    try:
        payload = jwt.decode(
            token,
            settings.JWT_SECRET,
            algorithms=[settings.JWT_ALGORITHM]
        )
        request.state.user_id = payload.get("sub")
    except JWTError:
        return JSONResponse(
            status_code=401,
            content={"detail": "Invalid token or expired token"}
        )

    return await call_next(request)
```

1.4 Technology Stack Details

Component	Technology	Version	Purpose	Rationale
Backend Framework	FastAPI	0.95.x	API development, request handling	High performance with async support, automatic OpenAPI docs, Pydantic validation
Frontend Framework	React.js	18.x	UI development	Component-based architecture, virtual DOM for performance, strong ecosystem
UI Components	Material-UI	5.x	Pre-designed components	Professional-looking UI with consistent design, responsive components
State Management	Redux Toolkit	1.9.x	Global state management	Predictable state updates, middleware support, developer tools
Primary Database	MongoDB	6.0	Document storage	Schema flexibility for rapidly evolving data models, JSON-like documents
Vector Database	Pinecone	2.x	Vector embeddings storage	Optimized for similarity search, scalable for large embedding collections
Caching	Redis	7.0	Caching, rate limiting	In-memory performance,

				pub/sub capabilities, TTL support
ML Orchestration	LangChain	0.1.x	LLM chains, RAG implementation	Composable components for LLM applications, abstracts provider-specific details
LLM Providers	OpenAI, Mistral, Hugging Face	Various	Natural language processing	Multiple providers for redundancy and specialized capabilities
Authentication	JWT	N/A	User authentication	Stateless authentication, suitable for distributed systems
Containerization	Docker	24.x	Application containerization	Consistent environments, isolates dependencies
Orchestration	Kubernetes	1.27	Container orchestration	Automated deployment, scaling, and management of containerized applications
CI/CD	GitHub Actions	N/A	Continuous integration/deployment	Tight GitHub integration, workflow automation
Monitoring	Prometheus & Grafana	2.x & 9.x	System monitoring	Time-series data collection, alerting, visualization

Technology Selection Criteria: Technologies were selected based on several factors including performance characteristics, ecosystem maturity, community support, and team expertise. Where possible, open-source technologies were preferred to avoid vendor lock-in, with the exception of certain LLM providers where proprietary solutions currently offer significant advantages in capability and cost-efficiency.