Robin AI Platform - Complete Documentation

Table of Contents

Project Overview

The Robin AI platform is a comprehensive stock analysis and trading platform that combines real-time market data analysis with advanced AI capabilities. The platform provides:

- Real-time market data streaming and analysis
- Advanced technical indicators and pattern recognition
- Options chain analysis and Greeks calculation
- AI-powered chat interface for market analysis
- Document-based knowledge system using RAG
- Secure and scalable infrastructure

The platform is built using a microservices architecture, with separate services for:

- Frontend (Next.js)
- Backend (FastAPI)
- Market data processing
- AI services (Ollama)
- Vector store (Pinecone)
- Caching (Redis)

Architecture

Microservices Architecture

1. **Frontend Service**
   - Next.js 14 application
   - TypeScript for type safety
   - Tailwind CSS for styling
   - WebSocket integration
   - Real-time updates
2. **Backend Service**
   - FastAPI application
   - REST and WebSocket endpoints
   - Authentication middleware
   - Rate limiting
   - Error handling
3. **Market Data Service**
   - Alpaca API integration
   - Real-time data streaming
   - Technical analysis

   - Options processing
   - Data caching

4. **AI Services**
   - Ollama containers
   - Multiple model support
   - Embedding generation
   - Response generation
   - Context management

5. **Data Services**
   - Pinecone vector store
   - Redis cache
   - File storage
   - Backup systems

Data Flow

1. **Real-time Data Flow**

   ```
   Market Data → Alpaca API → WebSocket Server → Frontend
   ↓
   Redis Cache
   ↓
   Analysis Engine
   ↓
   Response Generation
   ```

2. **Document Processing Flow**

   ```
   PDF Upload → Text Extraction → Cleaning → Chunking
   ↓
   Embedding Generation → Vector Store
   ↓
   Query Processing → Context Retrieval
   ↓
   Response Generation
   ```

Technical Stack

Frontend

- Next.js 14
- TypeScript
- Tailwind CSS
- WebSocket Client
- React Context
- Custom Hooks

Backend

- FastAPI
- Python 3.11
- Uvicorn
- Gunicorn
- Redis
- Pinecone

AI Services

- Ollama
- llama-embed-text
- llama2
- LangChain

Infrastructure

- Docker
- Docker Compose

- Nginx
- Redis
- Pinecone

System Components

Frontend Components

1. **Layout System**

   - Root layout (`layout.tsx`)
   - Page components
   - Navigation
   - Theme provider

2. **State Management**

   - React Context
   - Custom hooks
   - Local state
   - WebSocket state

3. **UI Components**

   - Chat interface
   - Market data display
   - Forms and inputs
   - Loading states

4. **Real-time Updates**

   - WebSocket connections
   - Market data streaming
   - Chat message updates
   - Error handling

Backend Components

1. **API Layer**

   - REST endpoints
   - WebSocket server
   - Authentication
   - Rate limiting

2. **Data Processing**

   - Market data analysis
   - Technical indicators
   - Options processing
   - Document processing

3. **AI Integration**

   - Embedding generation
   - Context retrieval
   - Response generation
   - Model management

4. **Storage Layer**

   - Vector store
   - Cache management
   - File storage
   - Backup systems

Data Flow

Real-time Data Processing

1. **Market Data Ingestion**

```python
class MarketStream:
    def __init__(self, api_key: str, secret_key: str):
        self.client = alpaca.Stream(api_key, secret_key)

    def subscribe(self, symbols: List[str]):
        """Subscribe to market data stream."""
        # Implementation details

    def handle_message(self, message: Dict):
        """Handle incoming market data."""
        # Implementation details
```

2. **Data Processing Pipeline**

```python
class DataProcessor:
    def process_market_data(self, data: MarketData):
        """Process market data."""
        # Implementation details

    def calculate_indicators(self, data: List[MarketData]):
        """Calculate technical indicators."""
        # Implementation details
```

3. **Real-time Updates**

```python
class WebSocketServer:
    def broadcast_update(self, data: Dict):
        """Broadcast market update to clients."""
        # Implementation details
```

Document Processing

1. **PDF Processing**

```python
class PDFCleaner:
    def extract_text_from_pdf(self, file_path: str) -> List[str]:
        """Extract text from PDF file."""
        # Implementation details

    def clean_text(self, text: str) -> str:
        """Clean extracted text."""
        # Implementation details
```

2. **Vector Store Integration**

```python
class VectorStore:
    def upsert_vectors(self, vectors: List[Dict]):
        """Upsert vectors to Pinecone."""
        # Implementation details

    def search_vectors(self, query_embedding: List[float]):
        """Search for similar vectors."""
        # Implementation details
```

Setup Instructions

Prerequisites

1. **System Requirements**
    - Docker and Docker Compose
    - Node.js 18+
    - Python 3.11+
    - Git
2. **API Keys**
    - Alpaca API key
    - Pinecone API key
    - OpenAI API key (optional)

Installation

1. **Clone Repository**

   ```
   git clone https://github.com/your-org/robin-ai.git
   cd robin-ai
   ```

2. **Environment Setup**

   ```
   # Create .env file
   cp .env.example .env
   # Edit .env with your API keys
   ```

3. **Start Services**

   ```
   docker-compose up -d
   ```

4. **Verify Installation**

   ```
   # Check service health
   curl http://localhost:8000/health
   ```

API Documentation

REST API

1. **Base URL**

   ```
   http://localhost:8000
   ```

2. **Authentication**

   ```
   Authorization: Bearer <token>
   ```

3. **Endpoints**

   - Health Check: `GET /health`
   - Stock Analysis: `POST /analyze`
   - Options Data: `GET /options/{symbol}`
   - Chat Interaction: `POST /chat`

WebSocket API

1. **Base URL**

   ```
   ws://localhost:8000
   ```

2. **Endpoints**

   - Market Data: `ws://localhost:8000/ws/market`
   - Chat Stream: `ws://localhost:8000/ws/chat`

Development Guide

Frontend Development

1. **Setup**

   ```
   cd frontend
   npm install
   npm run dev
   ```

2. **Component Structure**

   - Pages: `src/app/*`
   - Components: `src/components/*`
   - Styles: `src/styles/*`
   - Utils: `src/utils/*`

3. **State Management**

```
// Context example
const MarketContext = createContext<MarketContextType>({
  data: null,
  loading: false,
  error: null
});
```

Backend Development

1. **Setup**

```
cd backend
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

2. **API Development**

```
@app.post("/analyze")
async def analyze_stock(data: StockAnalysisRequest):
    """Analyze stock data."""
    # Implementation details
```

3. **Testing**

```
pytest tests/
```

Deployment Instructions

Production Deployment

1. **Build Process**

```
# Build Docker images
docker-compose build
```

2. **Environment Configuration**

```
# Production environment variables
NODE_ENV=production
API_URL=https://api.robin-ai.com
WS_URL=wss://api.robin-ai.com
```

3. **Service Deployment**

```
# Deploy services
docker-compose up -d
```

Scaling Strategy

1. **Horizontal Scaling**

```
services:
  backend:
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '2'
          memory: 4G
```

2. **Load Balancing**

```
upstream backend {
    server backend1:8000;
    server backend2:8000;
    server backend3:8000;
}
```

Troubleshooting Guide

Common Issues

1. **Service Health**

```
# Check service status
docker-compose ps
# View logs
docker-compose logs
```

2. **API Errors**

    - Check API keys
    - Verify service connectivity
    - Review error logs

3. **Performance Issues**

    - Monitor resource usage
    - Check cache hit rates
    - Review database queries

Health Checks

1. **Service Health**

```
@app.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "services": {
            "redis": check_redis(),
            "pinecone": check_pinecone(),
            "ollama": check_ollama()
        }
    }
```

2. **Performance Monitoring**

```
class PerformanceMonitor:
    def track_metrics(self):
        """Track performance metrics."""
        # Implementation details
```

Contributing Guidelines

Development Process

1. **Branch Strategy**

    - `main`: Production code
    - `develop`: Development branch
    - `feature/*`: Feature branches
    - `bugfix/*`: Bug fix branches

2. **Code Standards**

    - Follow PEP 8 for Python
    - Use ESLint for JavaScript
    - Write unit tests
    - Document changes

3. **Pull Request Process**

    - Create feature branch
    - Write tests
    - Update documentation
    - Submit PR for review

Code Review

1. **Review Checklist**

    - Code quality
    - Test coverage
    - Documentation
    - Performance impact
    - Security considerations

2. **Approval Process**

    - Two reviewers required
    - All tests must pass
    - Documentation updated
    - No security issues

File Structure

Project Organization

1. **Root Directory**

```
robin-ai/
├── frontend/
├── backend/
├── docs/
├── tests/
├── docker-compose.yml
└── README.md
```

2. **Frontend Structure**

```
frontend/
├── src/
│   ├── app/
│   ├── components/
│   ├── styles/
│   └── utils/
├── public/
└── package.json
```

3. **Backend Structure**

```
backend/
├── main.py
├── rag/
├── data/
├── tests/
└── requirements.txt
```

Frontend Documentation

Component Architecture

1. **Page Components**

```
// Home page
export default function Home() {
  const [symbol, setSymbol] = useState('');
  const [loading, setLoading] = useState(false);

  return (
    // JSX structure
  );
}
```

2. **UI Components**

```
// Chat interface
function ChatInterface() {
  const [messages, setMessages] = useState<Message[]>([]);
  const [input, setInput] = useState('');

  return (
    // JSX structure
  );
}
```

State Management

1. **Context Providers**

```
// Market context
const MarketContext = createContext<MarketContextType>({
  data: null,
  loading: false,
  error: null
});
```

2. **Custom Hooks**

```
// WebSocket hook
function useWebSocket(url: string) {
  const [data, setData] = useState(null);
  // Implementation details
}
```

Backend Documentation

API Implementation

1. **Route Definitions**

```
@app.post("/analyze")
async def analyze_stock(data: StockAnalysisRequest):
    """Analyze stock data."""
    # Implementation details
```

2. **WebSocket Server**

```
@app.websocket("/ws/market")
async def market_websocket(websocket: WebSocket):
    """Handle market data WebSocket."""
    # Implementation details
```

Data Processing

1. **Market Data**

```
class MarketProcessor:
    def process_data(self, data: MarketData):
        """Process market data."""
        # Implementation details
```

2. **Technical Analysis**

```
class TechnicalAnalysis:
    def calculate_indicators(self, data: List[MarketData]):
        """Calculate technical indicators."""
        # Implementation details
```

RAG System Documentation

Document Processing

1. **PDF Cleaner**

```python
class PDFCleaner:
    def extract_text_from_pdf(self, file_path: str) -> List[str]:
        """Extract text from PDF file."""
        # Implementation details

    def clean_text(self, text: str) -> str:
        """Clean extracted text."""
        # Implementation details
```

2. **Vector Store**

```python
class VectorStore:
    def upsert_vectors(self, vectors: List[Dict]):
        """Upsert vectors to Pinecone."""
        # Implementation details
```

Query Processing

1. **Context Retrieval**

```python
def search_context(query_embedding: List[float], k: int = 5) -> List[Dict]:
    """Search for relevant context."""
    # Implementation details
```

2. **Response Generation**

```python
def generate_response(query: str, context: List[Dict]) -> str:
    """Generate response with context."""
    # Implementation details
```

Market Data Processing

Data Models

1. **Market Data**

```python
class MarketData(BaseModel):
    symbol: str
    price: float
    volume: int
    timestamp: datetime
    indicators: Dict[str, float]
```

2. **Options Data**

```python
class OptionsData(BaseModel):
    symbol: str
    expiration: date
    strike: float
    type: str
    price: float
    volume: int
    open_interest: int
```

Processing Pipeline

1. **Data Ingestion**

```python
class MarketStream:
    def subscribe(self, symbols: List[str]):
        """Subscribe to market data stream."""
        # Implementation details
```

2. **Analysis Engine**

```python
class AnalysisEngine:
    def analyze_data(self, data: MarketData):
        """Analyze market data."""
        # Implementation details
```

Deployment and Infrastructure

Docker Configuration

1. **Frontend Service**

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 3000
CMD ["npm", "start"]
```

2. **Backend Service**

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["gunicorn", "main:app", "--workers", "4"]
```

Infrastructure Setup

1. **Service Configuration**

```
services:
  frontend:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NEXT_PUBLIC_API_URL=http://backend:8000
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - REDIS_URL=redis://redis:6379
```

2. **Monitoring Setup**

```
class Monitor:
    def track_metrics(self):
        """Track system metrics."""
        # Implementation details
```

[Continue with more detailed sections...]