Modular Agentic AI Trading Architecture — Full Project Structure

---

1. High-Level Components

Component	Role	Tech Suggestions
Agent Microservices	Each specialist AI model runs as an independent microservice exposing API endpoints	Python + FastAPI / Node.js + Express
Event Bus / Message Broker	Real-time asynchronous communication & event streaming between agents and controller	Redis Pub/Sub, RabbitMQ, or Kafka
Controller / Orchestrator	Aggregates agent outputs, compiles trading decisions, triggers execution	Python FastAPI or Node.js backend
Database	Logs all predictions, signals, agent decisions, and actual trade outcomes for analytics	PostgreSQL or TimescaleDB (time-series)
Frontend Dashboard	Real-time React app showing live signals, agent reasoning, chart overlays, and trade logs	React + TradingView Widget + WebSockets
Analytics & Reporting	Batch jobs or streaming jobs that analyze success rates, agent performance, risk metrics	Python Pandas, Jupyter, or BI tools

---

2. Agent Roles & APIs

Each agent is an autonomous microservice responsible for a specialist task:

Agent Name	Model Assigned	API Endpoint Example	Input	Output (JSON)
ChartAnalyst	Mistral Small 3.2	/agents/chartanalyst	Recent candle data, price history	Detected patterns, price zones, confidence scores

Agent Name	Model Assigned	API Endpoint Example	Input	Output (JSON)
RiskManager	Kimi K2	/agents/riskmanager	Signal details, portfolio info	Position size, stop-loss, risk/reward ratio
MarketSentinel	Qwen3 Coder	/agents/marketsentinel	Live tick data, volatility history	Volatility alerts, scalping zones
MacroForecaster	TNG Chimera	/agents/macroforecaster	News feed, event data	News impact, directional bias
TacticBot	GLM 4.5 Air / Horizon Alpha	/agents/tacticbot	Aggregated signals from other agents	Entry/exit ticks, breakout logic, trade instructions
PlatformPilot	Kimi Dev 72B	/agents/platformpilot	Trade decision confirmation	Logs signals, triggers alerts and automation on platform

---

3. Data Flow & Event Loop

Use an **event-driven architecture** with a central event bus (Redis Pub/Sub or Kafka):

- On each new candle or market event:
    1. Publish event to bus → ChartAnalyst, MarketSentinel, MacroForecaster consume.
    2. Their outputs are published back → RiskManager consumes.
    3. RiskManager output + prior agent outputs → TacticBot consumes.
    4. TacticBot confirms entry/exit → PlatformPilot triggers platform automation.
    5. PlatformPilot logs all details to the database and pushes updates to frontend.

This event flow corresponds to your Mermaid workflow diagram.

---

4. Database Design

Use **PostgreSQL** or a time-series optimized DB like **TimescaleDB** to handle trade logs, predictions, and outcome data.

Tables Overview:

Table	Description	Key Fields
agents	Registered agents and metadata	agent_id, name, model, status
trade_signals	Records each trading decision & signal	signal_id, timestamp, symbol, timeframe, agents_triggered (json array), signal_data (json)
trade_outcomes	Actual trade results for analytics	outcome_id, signal_id (FK), exit_price, exit_time, pnl, success_flag
macro_events	Store macro context linked to signals	event_id, signal_id (FK), event_name, forecast_bias, source

Every logged output follows the JSON schema you provided, serialized into the DB.

---

5. Backend Structure

Suggested repo/folder layout

bash

CopyEdit

/backend

├— /agents

│ ├— chartanalyst/

│ ├— riskmanager/

│ ├— marketsentinel/

│ ├— macroforecaster/

│ ├— tacticbot/

│ └— platformpilot/

```
├── /orchestrator
│   ├── event_bus.py     # handles Redis/Kafka pubsub
│   ├── decision_compiler.py  # aggregates agent outputs
│   └── api.py         # REST + WebSocket API for frontend
├── /db
│   ├── models.py       # SQLAlchemy or Prisma schemas
│   ├── migrations/
│   └── db_session.py
├── /utils
│   ├── logger.py
│   └── helpers.py
└── main.py            # Entry point for orchestrator service
```

---

6. Frontend Structure (React)

bash

CopyEdit

```
/frontend
├── /components
│   ├── LiveSignalFeed.js     # shows live signals, confidence
│   ├── ChartOverlay.js       # TradingView API integration
│   ├── AgentLogsPanel.js     # per-agent reasoning breakdown
│   ├── MacroEventFeed.js     # shows macro context events
│   └── TradeBook.js          # logs with PnL, success analytics
├── /services
│   ├── websocket.js          # connects to backend WebSocket API
│   └── api.js                # REST API calls
├── /utils
```

```
│   └── time.js            # time formatting helpers
└── App.js
```

- Use **WebSocket** for near-real-time feed updates.
- Integrate **TradingView Widget** or canvas overlays for chart annotations.

---

7. Agent Development Best Practices

- Agents run independently, exposing REST or RPC endpoints.
- Each agent:
    - Receives input JSON (e.g., candle data, market event).
    - Calls its assigned AI model internally.
    - Outputs JSON with its reasoning and signals.
    - Publishes results to the event bus.
- Use containerization (Docker) for isolation and deployment.
- Implement retries, rate-limiting, and logging for robustness.

---

8. Controller / Orchestrator Logic

- Listens to event bus channels for agent outputs.
- Aggregates signals from all relevant agents per event.
- Applies business logic/rules to confirm final trade signals.
- Calls PlatformPilot for logging and execution triggers.
- Emits final trade signals and agent reasoning to frontend via WebSocket.

---

9. Logging & Outcome Tracking

- Every trade decision is logged immediately.
- When trades close, the outcome (profit/loss, exit price/time) is pushed back into the DB.
- Build a **periodic analytics job** that computes:
    - Agent accuracy (predictions vs. outcomes).

    - Risk-adjusted returns.
    - Success rate per pattern or news event.
- Analytics help improve agent tuning and strategy adjustments.

---

10. Technology Stack Summary

Layer	Technology
AI Models	Mistral, Kimi K2, Qwen3, GLM, TNG Chimera, Horizon Alpha
Backend	Python (FastAPI), Redis / Kafka, PostgreSQL / TimescaleDB
Frontend	React, TradingView Widget, WebSockets
DevOps	Docker, Kubernetes (optional), CI/CD pipelines
Analytics	Python Pandas, Jupyter, or BI tools (Metabase, Superset)