Starter Repo Layout — Modular Agentic AI Trading System

Repo structure

```
pgsql
CopyEdit
agentic-trading/
├── backend/
│   ├── agents/
│   │   ├── chartanalyst/
│   │   │   ├── main.py
│   │   │   ├── model.py
│   │   │   └── utils.py
│   │   ├── riskmanager/
│   │   │   ├── main.py
│   │   │   ├── model.py
│   │   │   └── utils.py
│   │   ├── marketsentinel/
│   │   │   ├── main.py
│   │   │   ├── model.py
│   │   │   └── utils.py
│   │   ├── macroforecaster/
│   │   │   ├── main.py
│   │   │   ├── model.py
│   │   │   └── utils.py
│   │   ├── tacticbot/
│   │   │   ├── main.py
│   │   │   ├── model.py
│   │   │   └── utils.py
│   │   └── platformpilot/
│   │       ├── main.py
│   │       ├── model.py
│   │       ├── utils.py
│   │       └── automation.py   # clicking/link automation here
│   ├── orchestrator/
│   │   ├── event_bus.py         # Redis pubsub wrapper
│   │   ├── decision_compiler.py # Aggregates agent outputs
│   │   ├── api.py               # FastAPI REST + WebSocket API
│   │   ├── models.py            # DB schemas (SQLAlchemy)
│   │   └── main.py              # Orchestrator startup
│   ├── db/
│   │   ├── migrations/
│   │   └── db_session.py
│   ├── config.py                # Config & API keys (Travily, Redis, DB)
│   └── requirements.txt
├── frontend/
│   ├── public/
│   ├── src/
│   │   ├── components/
│   │   │   ├── LiveSignalFeed.js
│   │   │   ├── ChartOverlay.js
│   │   │   ├── AgentLogsPanel.js
│   │   │   ├── MacroEventFeed.js
```

```
│   │   │   └── TradeBook.js
│   │   ├── services/
│   │   │   ├── websocket.js
│   │   │   └── api.js
│   │   ├── utils/
│   │   │   └── time.js
│   │   └── App.js
│   ├── package.json
│   └── vite.config.js
├── docker-compose.yml
└── README.md
```

Backend Details

1. Agent Stub (example: `chartanalyst/main.py`)

```
python
CopyEdit
from fastapi import FastAPI, Request
import uvicorn

app = FastAPI(title="ChartAnalyst Agent")

@app.post("/detect_pattern")
async def detect_pattern(payload: dict):
    # TODO: integrate with Mistral Small 3.2 or mock output for now
    candles = payload.get("candles", [])
    # Stubbed response
    response = {
        "pattern": "Bullish Engulfing",
        "confidence": 0.85,
        "price_zones": {"support": 1970, "resistance": 1980}
    }
    return response

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8001)
```

Each agent follows this lightweight FastAPI microservice pattern, just change ports and endpoints accordingly.

---

2. Event Bus Wrapper (`orchestrator/event_bus.py`)

```
python
CopyEdit
import asyncio
import aioredis
import json

class RedisPubSub:
    def __init__(self, redis_url="redis://localhost"):
        self.redis_url = redis_url
        self.redis = None
        self.pub = None
```

```
        self.sub = None

    async def connect(self):
        self.redis = await aioredis.from_url(self.redis_url)
        self.pub = self.redis.pubsub()
        self.sub = self.redis.pubsub()

    async def publish(self, channel: str, message: dict):
        await self.redis.publish(channel, json.dumps(message))

    async def subscribe(self, channel: str):
        await self.sub.subscribe(channel)

    async def listen(self):
        async for message in self.sub.listen():
            if message['type'] == 'message':
                data = json.loads(message['data'])
                yield data
```

---

3. Orchestrator API + WebSocket (`orchestrator/api.py`)

- REST endpoints for frontend data queries
- WebSocket to push live agent signal events

Basic WebSocket example with FastAPI:

```
python
CopyEdit
from fastapi import FastAPI, WebSocket
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

clients = []

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    clients.append(websocket)
    try:
        while True:
            data = await websocket.receive_text()  # or receive_json()
            # Echo or handle client messages if needed
    except Exception:
        clients.remove(websocket)

async def broadcast(message: dict):
    living_clients = []
    for client in clients:
        try:
            await client.send_json(message)
```

```
            living_clients.append(client)
        except Exception:
            pass
    clients[:] = living_clients
```

---

4. Database Models (`orchestrator/models.py`)

Using SQLAlchemy:

```
python
CopyEdit
from sqlalchemy import Column, Integer, String, DateTime, Float, JSON,
ForeignKey, Boolean
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.sql import func

Base = declarative_base()

class TradeSignal(Base):
    __tablename__ = "trade_signals"
    signal_id = Column(Integer, primary_key=True, index=True)
    timestamp = Column(DateTime(timezone=True), server_default=func.now())
    symbol = Column(String)
    timeframe = Column(String)
    agents_triggered = Column(JSON)
    signal_data = Column(JSON)
    macro_context = Column(JSON)
    logged_by = Column(String)

class TradeOutcome(Base):
    __tablename__ = "trade_outcomes"
    outcome_id = Column(Integer, primary_key=True, index=True)
    signal_id = Column(Integer, ForeignKey("trade_signals.signal_id"))
    exit_price = Column(Float)
    exit_time = Column(DateTime(timezone=True))
    pnl = Column(Float)
    success_flag = Column(Boolean)
```

---

5. Automation Module (`platformpilot/automation.py`)

Automating clicking and link following post web search:

```
python
CopyEdit
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options
import time

class WebAutomation:
    def __init__(self, headless=True):
        options = Options()
        options.headless = headless
        self.driver = webdriver.Chrome(options=options)

    def search_and_click(self, search_url: str, link_text: str):
```

```
        self.driver.get(search_url)
        time.sleep(3)  # wait for page load (replace with smarter wait)

        try:
            link = self.driver.find_element(By.LINK_TEXT, link_text)
            link.click()
            return True
        except Exception as e:
            print(f"Link not found or error: {e}")
            return False

    def close(self):
        self.driver.quit()
```

You can call this after a web search result, for example triggered by MacroForecaster or MarketSentinel.

---

6. Travily API Integration (`agents/macroforecaster/utils.py`)

Example wrapper:

```
python
CopyEdit
import requests

class TravilyClient:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.travily.io/v1"

    def get_latest_events(self, market="forex"):
        headers = {"Authorization": f"Bearer {self.api_key}"}
        resp = requests.get(f"{self.base_url}/events?market={market}",
headers=headers)
        if resp.status_code == 200:
            return resp.json()
        else:
            return {}
```

Frontend Basic Example

- React app with WebSocket live feed connection in `/frontend/src/services/websocket.js`

```
javascript
CopyEdit
export class SignalWebSocket {
  constructor(url) {
    this.ws = new WebSocket(url);
    this.ws.onopen = () => console.log("WebSocket connected");
    this.ws.onclose = () => console.log("WebSocket disconnected");
  }

  onMessage(callback) {
```

```
    this.ws.onmessage = (event) => {
      const data = JSON.parse(event.data);
      callback(data);
    };
  }

  send(data) {
    this.ws.send(JSON.stringify(data));
  }
}
```

- React component example `/frontend/src/components/LiveSignalFeed.js` to show incoming signals.

Docker Compose (simplified for dev)

```
yaml
CopyEdit
version: "3.8"
services:
  redis:
    image: redis:7
    ports:
      - "6379:6379"

  orchestrator:
    build: ./backend/orchestrator
    ports:
      - "8000:8000"
    depends_on:
      - redis

  chartanalyst:
    build: ./backend/agents/chartanalyst
    ports:
      - "8001:8001"
    depends_on:
      - redis

  # similarly for other agents...
```

Summary

- Modular FastAPI microservices for agents, each exposing clear endpoints.
- Redis Pub/Sub for event-driven agent communication.
- Orchestrator aggregates & compiles decisions, exposes REST + WebSocket APIs.
- PostgreSQL + SQLAlchemy for logging signals and outcomes.
- Frontend React dashboard with WebSocket live feed + TradingView overlay.
- Selenium automation for post-search link clicking (in `platformpilot`).
- Travily API wrapper for enriched macro events.