



Kiến trúc cho Đồ án Hệ thống Phân tích và Dự báo Xu hướng Tài chính AI

Kiến trúc có thể áp dụng cho đồ án hệ thống phân tích và dự báo xu hướng tài chính AI. Mỗi kiến trúc được phân tích theo tính phù hợp với yêu cầu (tách rời mối quan tâm, khả năng mở rộng, xử lý thời gian thực, tích hợp AI).

1. Microservices (Kiến trúc dịch vụ nhỏ)

Mô tả

Kiến trúc microservices phân rã hệ thống thành các dịch vụ nhỏ, tự vận hành. Mỗi dịch vụ giải quyết một chức năng nghiệp vụ, có thể phát triển, triển khai và mở rộng độc lập. Kiến trúc này mang lại tính phân quyền (decentralization) và khả năng chịu lỗi, bởi khi một dịch vụ gặp sự cố, những dịch vụ khác vẫn hoạt động.

Ứng dụng trong đồ án:

1

Real-time Charting Service

cung cấp biểu đồ giá thời gian thực cho từng mã (BTCUSDT, ETHUSDT).

2

Data Collector Service

thu thập dữ liệu từ API của Binance, normalise và đẩy vào message broker.

3

News Crawler & Sentiment Analysis

các microservice riêng biệt để thu thập và phân tích tin tức.

4

AI Prediction Service

microservice chứa mô hình dự đoán giá; sử dụng giao diện API để nhận dữ liệu đầu vào và trả kết quả.

5

Backtesting Service

thực hiện mô phỏng chiến lược đầu tư trên dữ liệu lịch sử.

6

Auth Service

quản lý xác thực, tạo JWT và ủy quyền người dùng.

Thiết kế này tuân theo nguyên tắc tách biệt mối quan tâm và dễ mở rộng.

2. Containers (Đóng gói bằng container)

Mô tả

Container hóa gói ứng dụng và tất cả phụ thuộc thành một đơn vị độc lập, giúp chạy nhất quán trên các môi trường khác nhau. Containers chia sẻ nhân hệ điều hành nên nhẹ hơn VM, khởi động nhanh và có khả năng cách ly cao statsig.com. Điều này cải thiện tính di động ("write once, run anywhere") và hiệu quả tài nguyên.

Ứng dụng:

Containerization

Mỗi microservice nên được đóng gói thành một container (dùng Docker). Ví dụ: gói dịch vụ Dự đoán giá (AI Prediction Service) với môi trường Python và thư viện học máy; gói News Crawler Service với môi trường Node/Python và thư viện scraping.

Triển khai

Containers cho phép triển khai nhất quán lên cụm Kubernetes, AWS ECS hoặc nền tảng serverless container khác.

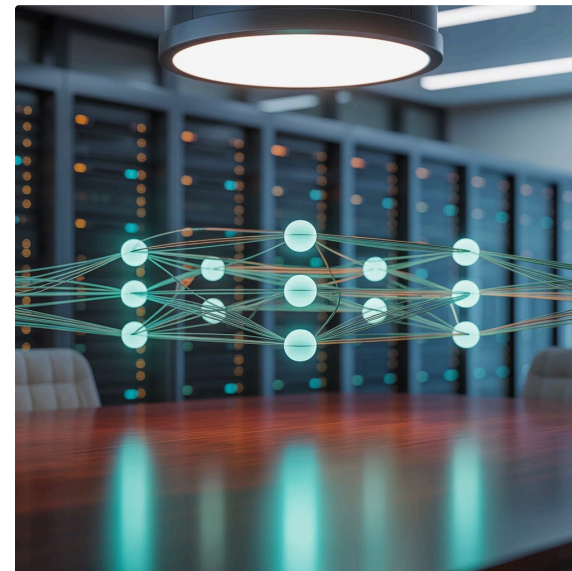
Bảo mật

Ngoài ra, container còn tăng cường bảo mật do cách ly tiến trình.

3. Service Mesh

Mô tả

Khi số lượng microservice tăng, việc quản lý giao tiếp giữa chúng trở nên phức tạp. Service mesh là lớp hạ tầng chuyên dụng quản lý và bảo vệ giao tiếp giữa các microservice, cung cấp tính năng như cân bằng tải, phát hiện dịch vụ, mã hóa mTLS, retry, observability và chính sách traffic. Triển khai điển hình dựa trên proxy sidecar (Linkerd, Istio) được gắn kèm mỗi container.



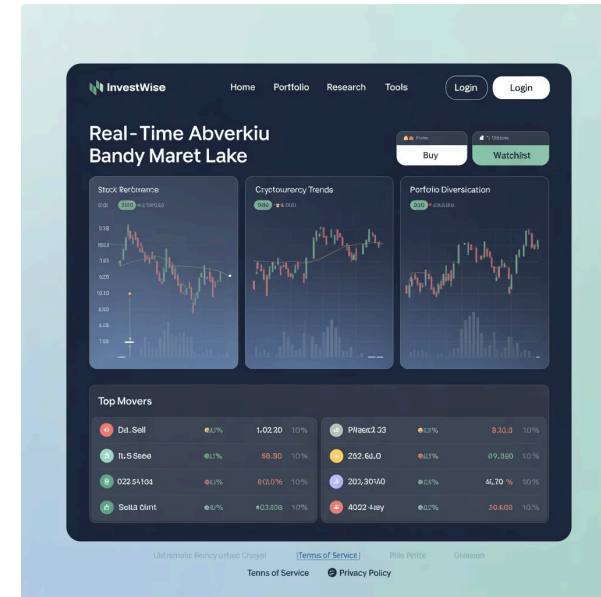
Ứng dụng:

Với đồ án này, service mesh rất hữu ích khi có nhiều microservice và cần theo dõi, bảo mật kết nối. Ví dụ: khi AI Prediction Service gọi Historical Data Warehouse qua API, service mesh đảm bảo chứng thực mTLS, tự động retry nếu lỗi, và cung cấp metrics/trace để phát hiện vấn đề. Tuy nhiên, nếu quy mô còn nhỏ (<5–6 dịch vụ), có thể trì hoãn service mesh để giảm phức tạp.

4. SPA Framework (Kiến trúc Single-Page Application cho giao diện)

Mô tả

Single-page application (SPA) là trang web chỉ tải một tài liệu HTML và cập nhật nội dung động mà không cần reload toàn bộ trang. Nhờ chỉ cập nhật phần dữ liệu cần thiết, SPA mang lại tốc độ tải nhanh, trải nghiệm người dùng liền mạch và tương tác thời gian thực.



Ứng dụng:

Giao diện người dùng của hệ thống (dashboard) nên được xây dựng bằng SPA framework như React, Vue hoặc Angular. Ví dụ:



Real-time Charting UI

sử dụng React + D3.js để hiển thị biểu đồ giá cập nhật liên tục.



News Feed UI

hiển thị tin tức và cảm xúc phân tích dưới dạng luồng.



Backtesting UI

cho phép người dùng cấu hình chiến lược và xem kết quả mô phỏng. SPA cho phép kết nối qua WebSocket để nhận dữ liệu thời gian thực và cập nhật ngay trên trang, tránh tải lại toàn bộ trang.

5. Application Frameworks

Mô tả:

Framework backend cung cấp bộ công cụ và cấu trúc mẫu để xây dựng ứng dụng, bao gồm routing, authentication, kết nối cơ sở dữ liệu, v.v. Một framework tốt phải cung cấp các tác vụ tiêu chuẩn như routing REST API, xác thực, thực thi logic nghiệp vụ, truy xuất cơ sở dữ liệu và tích hợp với microservices khác hostman.com.

Ứng dụng:

Tùy theo ngôn ngữ, có thể chọn:

Spring Boot (Java) / ASP.NET Core

cho các dịch vụ cần hiệu suất cao và cấu trúc nghiêm ngặt. Ví dụ: dịch vụ Backtesting hay Auth Service có thể dùng Spring Boot để tận dụng hỗ trợ tốt cho JWT, OAuth2 và cấu hình bảo mật.

Django hoặc FastAPI (Python)

cho dịch vụ AI Prediction; giúp xây dựng API nhanh với ORM và support tốt cho thư viện học máy hostman.com.

Express (Node.js)

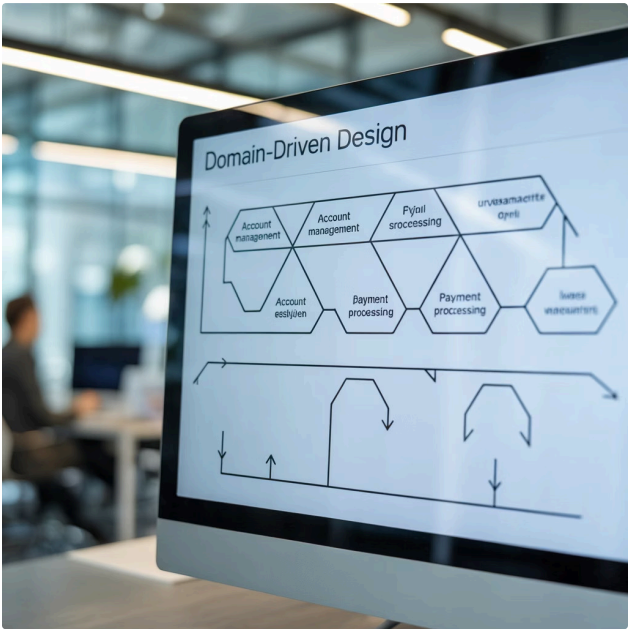
cho News Crawler Service hoặc WebSocket Gateway vì Node có nhiều thư viện hỗ trợ streaming và websockets.

Frameworks giúp tăng tốc phát triển, cung cấp công cụ testing, và dễ tích hợp vào CI/CD.

6. Domain-Driven Design (DDD)

Mô tả

DDD tập trung vào mô hình hóa domain nghiệp vụ; microservice được thiết kế xoay quanh các nghiệp vụ chuyên môn (business capabilities) và có ranh giới (bounded context) rõ ràng. Bằng cách phân tích domain, ta xác định các bounded context và tách dịch vụ sao cho mỗi dịch vụ có trách nhiệm duy nhất. DDD gồm hai giai đoạn: chiến lược (xác định ranh giới) và chiến thuật (sử dụng các pattern như Entities, Aggregates, Domain Services).



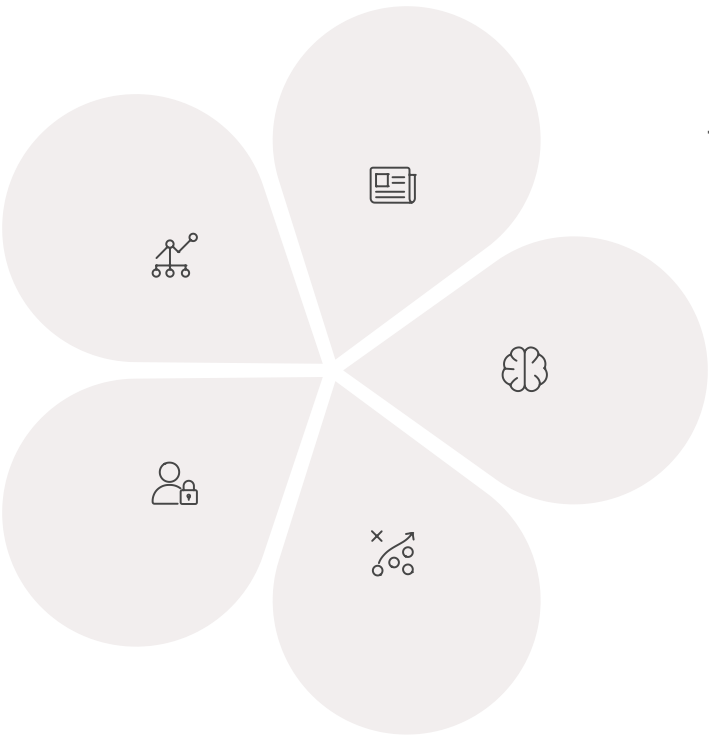
Ứng dụng:

Domain "Market Data"

xử lý dữ liệu giá, bao gồm Data Collector và Real-time Charting.

Domain "Account Management"

xác thực và quyền truy cập người dùng.



Domain "News & Sentiment"

thu thập và phân tích tin tức.

Domain "Prediction Models"

huấn luyện và suy luận mô hình AI.

Domain "Investment Strategies"

backtesting và tối ưu hóa chiến lược.

Thiết kế DDD giúp xác định ranh giới microservice chính xác và giảm coupling.

Domain 1: Market Data

- **Nhiệm vụ:** Thu thập giá từ Binance, hiển thị biểu đồ.
- **Bounded Context:**
 - Entity: MarketInstrument (BTCUSDT, ETHUSDT).
 - Value Object: PriceTick (giá, thời gian, khối lượng).
 - Domain Service: "Tính trung bình giá 24h".

Domain 2: News & Sentiment

- **Nhiệm vụ:** Crawl tin tức, phân tích cảm xúc.
- **Bounded Context:**
 - Entity: NewsArticle.
 - Value Object: SentimentScore (từ -1 đến 1).
 - Domain Service: "Phân tích cảm xúc từ tiêu đề và nội dung".

Domain 3: Prediction Models

- **Nhiệm vụ:** Huấn luyện và dự đoán giá.
- **Bounded Context:**
 - Entity: PredictionModel (phiên bản mô hình).
 - Value Object: PredictionResult.
 - Domain Service: "Chạy dự đoán giá dựa trên dữ liệu 1h gần nhất".

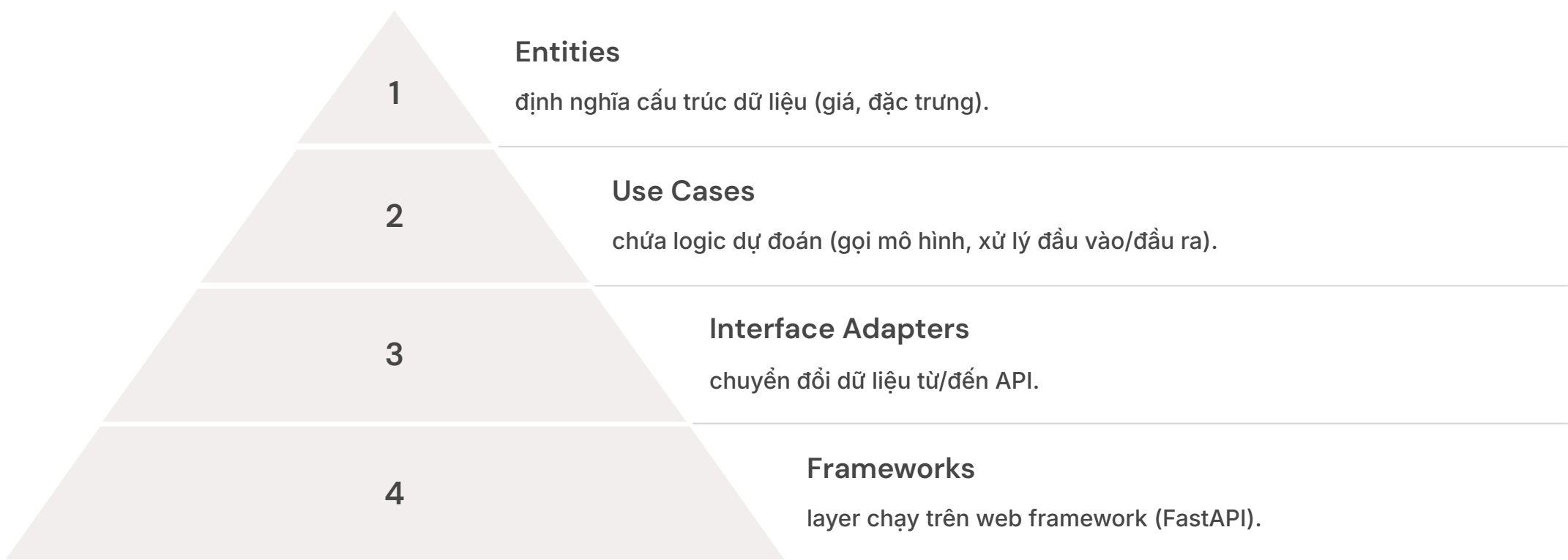
7. Clean Architecture

Mô tả

Clean Architecture (Robert C. Martin) phân chia phần mềm thành các lớp độc lập (Entities, Use Cases, Interface Adapters, Frameworks). Mục tiêu là giữ logic nghiệp vụ độc lập với framework, UI và cơ sở dữ liệu. Những lợi ích chính gồm tính bảo trì, khả năng kiểm thử cao, linh hoạt thay thế công nghệ và tính mở rộng.

Ứng dụng:

Trong mỗi microservice, áp dụng Clean Architecture sẽ làm cho mô hình AI hay logic backtesting tách biệt khỏi lớp hạ tầng. Ví dụ trong AI Prediction Service:



Cách tổ chức này giúp dễ thay đổi mô hình AI mà không ảnh hưởng đến API.

a) AI Prediction Service

- **Entities:** PredictionModel, PredictionResult (các quy tắc tính toán kết quả dự đoán).
- **Use Cases:** "Dự đoán giá BTCUSDT trong 1h tới dựa trên dữ liệu kline".
- **Interface Adapters:** Repository interface để lấy dữ liệu từ message broker, chuyển đổi thành PriceTick list cho use case.
- **Frameworks & Drivers:** FastAPI (API HTTP), PyTorch (chạy mô hình), Kafka client (nhận dữ liệu).

b) News & Sentiment Service

- **Entities:** NewsArticle, SentimentScore.
- **Use Cases:** "Phân tích cảm xúc từ danh sách tin tức mới nhận".
- **Interface Adapters:** Adapter để gọi mô hình NLP, mapper để chuyển kết quả thành dạng lưu vào DB.
- **Frameworks & Drivers:** Scrapy/Playwright (crawling), PostgreSQL, TensorFlow.

c) Account Management Service

- **Entities:** UserAccount, UserRole.
- **Use Cases:** "Đăng nhập và tạo JWT token".
- **Interface Adapters:** Adapter kiểm tra thông tin trong DB, mapper dữ liệu sang payload JWT.
- **Frameworks & Drivers:** Spring Boot, MySQL, OAuth2/JWT library.

8. Transactional Processing – SAGA Pattern

Mô tả

Trong kiến trúc microservices, mỗi dịch vụ có cơ sở dữ liệu riêng. Để xử lý các giao dịch xuyên nhiều dịch vụ (ví dụ: ghi nhận lệnh và trừ tiền tài khoản), không thể dùng transaction ACID truyền thống; thay vào đó, sử dụng Saga. Saga là chuỗi các giao dịch cục bộ; mỗi bước cập nhật DB và phát sự kiện để kích hoạt bước tiếp theo. Nếu một bước thất bại, thực thi các giao dịch bù (compensating transactions). Có hai kiểu phối hợp: choreography (các dịch vụ tự phát sự kiện) và orchestration (một orchestrator điều khiển thứ tự).



Ứng dụng:

Ví dụ: khi người dùng mua gói dịch vụ cao cấp, hệ thống phải ghi nhận thanh toán, cập nhật quyền truy cập và gửi biên nhận:



Order Service

tạo record đơn hàng rồi phát sự kiện OrderCreated.



Payment Service

xử lý thanh toán; nếu thành công, phát sự kiện PaymentConfirmed.



Auth Service

nâng cấp quyền người dùng; nếu thất bại, phát sự kiện PaymentFailed và Order Service thực hiện giao dịch bù (hủy đơn). Saga giúp đảm bảo nhất quán mà không cần khóa dữ liệu trên nhiều dịch vụ.

Choreography – “Tự phát sự kiện”

Bản chất:

- Không có “người chỉ huy” trung tâm.
- Mỗi dịch vụ **tự biết** khi nào cần hành động bằng cách **lắng nghe** các sự kiện từ dịch vụ khác và **tự phát** sự kiện mới.
- Giống như một nhóm nhảy mà ai cũng biết động tác tiếp theo khi nghe nhạc, không cần người hô nhịp.

Ví dụ trong hệ thống bạn:

Người dùng mua gói dịch vụ cao cấp:

1. **Account Service** tạo yêu cầu mua → **phát sự kiện** `PurchaseRequested`.
2. **Payment Service** **lắng nghe** `PurchaseRequested` → xử lý thanh toán → nếu thành công, **phát** `PaymentConfirmed`.
3. **Subscription Service** **lắng nghe** `PaymentConfirmed` → nâng cấp quyền → **phát** `SubscriptionUpgraded`.
4. **Notification Service** **lắng nghe** `SubscriptionUpgraded` → gửi email thông báo.

Điểm mạnh:

- Đơn giản, không cần một thành phần trung tâm.
- Dễ mở rộng: chỉ cần thêm dịch vụ mới lắng nghe sự kiện là xong.

Điểm yếu:

- Khó quản lý luồng tổng thể khi số bước và số dịch vụ nhiều.
- Dễ bị “event spaghetti” (chuỗi sự kiện chồng chéo khó theo dõi).

Orchestration – “Nhạc trưởng” điều khiển dàn nhạc

Bản chất:

- Có một dịch vụ trung tâm gọi là **Orchestrator**.
- Orchestrator **ra lệnh** cho từng dịch vụ thực hiện công việc theo thứ tự, dựa trên kết quả của bước trước.
- Giống như một nhạc trưởng điều khiển dàn nhạc, ra hiệu cho từng nhạc cụ chơi đúng lúc.

Ví dụ trong hệ thống bạn:

Người dùng mua gói dịch vụ cao cấp:

1. **Orchestrator Service** nhận yêu cầu mua → gọi **Account Service** xác nhận thông tin người dùng.
2. Nếu OK, Orchestrator gọi **Payment Service** để xử lý thanh toán.
3. Nếu thanh toán thành công, Orchestrator gọi **Subscription Service** để nâng cấp quyền.
4. Sau đó, Orchestrator gọi **Notification Service** để gửi email xác nhận.

Điểm mạnh:

- Dễ quản lý và theo dõi toàn bộ quy trình.
- Xử lý lỗi, rollback dễ dàng hơn (Orchestrator biết bước nào thất bại và gọi bù).

Điểm yếu:

- Orchestrator có thể trở thành “điểm lỗi đơn” (single point of failure).
- Có nguy cơ bị “quá tải” nếu mọi luồng giao dịch đều qua Orchestrator.

9. CQRS (Command/Query Responsibility Segregation) và Event Sourcing

Mô tả CQRS

CQRS tách biệt mô hình đọc và mô hình ghi. Các lệnh (commands) cập nhật dữ liệu, còn các truy vấn (queries) lấy dữ liệu. Mô hình ghi và mô hình đọc có thể tối ưu riêng, giúp cải thiện hiệu suất và khả năng mở rộng. Điều này đặc biệt hữu ích khi tần suất đọc rất cao và cấu trúc dữ liệu cho đọc khác với dữ liệu ghi.

Ứng dụng:

Backtesting Service với CQRS

phần ghi (command) lưu chiến lược và kết quả, phần đọc sử dụng cơ sở dữ liệu tối ưu cho phân tích (ví dụ ClickHouse) để hiển thị biểu đồ.

AI Prediction Service

khi cập nhật mô hình mới, ta ghi một sự kiện "ModelUpdated"; sự kiện này cập nhật cache và trigger retraining pipeline.

Event Sourcing cho lịch sử giao dịch

Lưu trạng thái chiến lược/dữ liệu giao dịch bằng Event Sourcing giúp tái tạo lịch sử và phát hiện lỗi dễ dàng; ví dụ: mỗi giao dịch được lưu thành sự kiện TransactionCreated, TransactionExecuted, TransactionFailed.

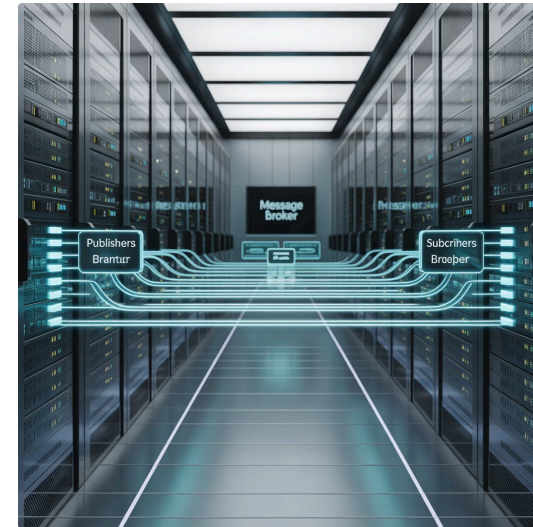
Mô tả Event Sourcing

Thay vì lưu trạng thái hiện tại, Event Sourcing lưu trữ toàn bộ các sự kiện thay đổi trạng thái; việc tái tạo trạng thái bằng cách phát lại chuỗi sự kiện. Event store cũng có thể đóng vai trò message broker khi dịch vụ khác subscribe sự kiện. Lợi ích bao gồm khả năng audit 100% lịch sử, dễ dàng phát lại và đảm bảo sự kiện không mất.

10. Message Broker / Message Bus

Mô tả

Message broker là hạ tầng trung gian giúp các dịch vụ giao tiếp bất đồng bộ. Một message bus cung cấp mô hình dữ liệu chung, tập lệnh chung và hạ tầng messaging để các ứng dụng giao tiếp thông qua giao diện chung. Các broker phổ biến: Apache Kafka, RabbitMQ, Redis Streams. Broker giúp giảm coupling, hỗ trợ pub/sub, độ bền dữ liệu và cân bằng tải.



Ứng dụng:

Hệ thống cần một message broker để truyền dữ liệu thời gian thực và sự kiện:



Stream dữ liệu giá

Data Collector publish dữ liệu giá vào topic (ví dụ "binance.btcusdt.kline.1m").
Real-time Charting Service và AI Prediction Service subscribe.



Event Sourcing & Saga

các dịch vụ publish sự kiện (OrderCreated, PaymentConfirmed, v.v.) và subscribe để thực thi logic.



Broadcast thông báo

khi mô hình AI được cập nhật, publish sự kiện để client cache reload.

Luồng dữ liệu giá thời gian thực (Pub/Sub)

1. **Data Collector Service** lấy dữ liệu giá từ Binance → **Publish** lên topic `binance.btcusdt.kline.1m` trong Kafka.
2. **Real-time Charting Service** **Subscribe** topic này → hiển thị biểu đồ.
3. **AI Prediction Service** cũng **Subscribe** cùng topic để chạy dự đoán.
4. Nếu sau này thêm **Alert Service** để gửi thông báo khi giá biến động >5%, chỉ cần subscribe topic đó mà không chạm vào code các dịch vụ khác.

Luồng tin tức & cảm xúc (Queue + Pub/Sub kết hợp)

1. **News Crawler Service** thu thập tin tức → gửi vào hàng đợi `news-raw`.
2. **Sentiment Analysis Service** lấy tin từ queue, phân tích, rồi publish kết quả vào topic `news.sentiment`.
3. **Dashboard Service** subscribe `news.sentiment` để hiển thị cho người dùng.

11. Event Streaming & Event-Driven Architecture (EDA)

Mô tả

EDA cho phép hệ thống phát hiện và phản ứng với sự kiện theo thời gian thực; các thành phần tách rời và giao tiếp bất đồng bộ [confluent.io](#). EDA thường kết hợp microservices và công nghệ event streaming như Apache Kafka để xử lý và phân phối lượng lớn sự kiện. Ưu điểm gồm độ trễ thấp, khả năng mở rộng, tách lòng và linh hoạt.



Ứng dụng:

Dữ liệu giá tài chính và tin tức có tính thời gian thực cao, nên hệ thống nên sử dụng EDA:

Data Collector

publish sự kiện giá; Charting và AI Prediction Service xử lý sự kiện ngay khi nhận được.

Backtesting Service

có thể publish kết quả; UI Service subscribe để cập nhật ngay lập tức. EDA giúp hệ thống phản ứng nhanh với biến động thị trường và cung cấp trải nghiệm thời gian thực cho người dùng.

1

2

3

News Crawler

publish sự kiện "NewsFetched"; Sentiment Analysis Service tiêu thụ và phân tích.




12. Serverless Architecture

Mô tả

Serverless là mô hình điện toán đám mây nơi nhà cung cấp quản lý hạ tầng và tự động scale. Code được triển khai thành các hàm stateless (function) và chạy khi có sự kiện; tính phí chỉ theo thời gian chạy. Serverless giúp rút ngắn thời gian triển khai và linh hoạt mở rộng.

Ứng dụng:

Một số thành phần có thể triển khai dưới dạng serverless:

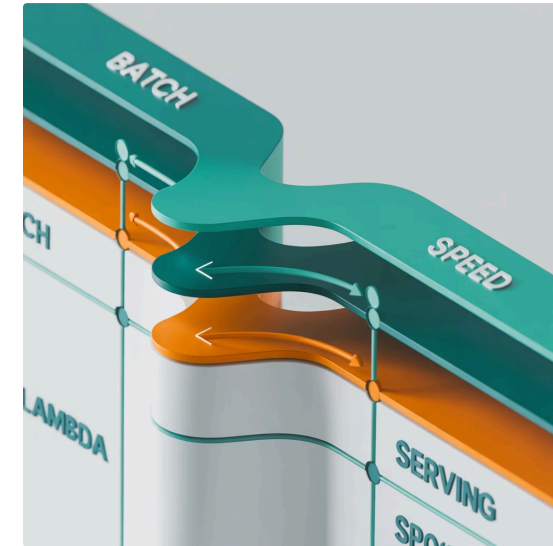
	Notification Function gửi email/SMS khi có cảnh báo giao dịch.		Periodic Tasks trigger pipeline retrain mô hình hàng tuần hoặc thu thập dữ liệu bổ sung.		Sentiment Analysis nếu khối lượng tin tức biến động và không liên tục, có thể sử dụng serverless function để phân tích khi có tin mới.
---	--	---	--	---	--

Tuy nhiên, các dịch vụ có kết nối WebSocket lâu dài (Real-time Charting) và xử lý nặng (Backtesting) nên chạy trên container/kubernetes thay vì serverless để tránh chi phí cao và giới hạn thời gian chạy.

13. Lambda Architecture

Mô tả

Lambda Architecture kết hợp batch layer, serving layer và speed layer để xử lý đồng thời dữ liệu lịch sử và dữ liệu thời gian thực. Dữ liệu từ nguồn (thường qua Kafka) được gửi song song tới batch layer để lưu trữ toàn bộ lịch sử và serving layer để lập chỉ mục, và tới speed layer để lập chỉ mục nhanh những dữ liệu mới. Tầng speed xử lý dữ liệu mới bằng stream processing nhằm giảm độ trễ, trong khi batch layer đảm bảo tính nhất quán.



Ứng dụng:

Nếu hệ thống cần phân tích trên cả dữ liệu lịch sử dài hạn và dữ liệu thời gian thực (ví dụ: huấn luyện mô hình AI trên lịch sử 5 năm và dự báo trên dữ liệu từng phút), có thể áp dụng Lambda Architecture:

Batch Layer

sử dụng Hadoop/Spark để lưu trữ và xử lý toàn bộ kline lịch sử và tin tức.

Serving Layer

xây dựng chỉ mục (OLAP) để trả lời truy vấn lịch sử.

Speed Layer

sử dụng Apache Flink hoặc Kafka Streams để phân tích dữ liệu giá mới trong vài giây.

Việc kết hợp hai layer giúp giảm độ trễ và vẫn đảm bảo dữ liệu đầy đủ hazelcast.com. Tuy nhiên, kiến trúc này phức tạp và chỉ nên áp dụng khi có nhu cầu xử lý Big Data quy mô lớn; đối với đồ án mang tính học thuật, cấu trúc này có thể phức tạp hơn cần thiết.

14. Software Integration & Message Bus

Mô tả

Phân tán hệ thống đòi hỏi tích hợp giữa các ứng dụng, dữ liệu, dịch vụ và thiết bị. Integration architecture cung cấp các công cụ như API Gateway, message queues, event grid và workflow để kết nối các ứng dụng và orchestrate quy trình. Message Bus là kiến trúc cho phép các ứng dụng hoạt động cùng nhau theo cách decoupled, sử dụng mô hình dữ liệu và command chung.



Ứng dụng:



API Gateway

làm cổng vào duy nhất; xác thực JWT, giới hạn tốc độ và chuyển tiếp tới microservice.



Workflow Orchestration

dùng Apache Airflow hoặc Azure Logic Apps để orchestrate pipeline backtesting.



Message Bus

sử dụng Kafka hoặc RabbitMQ làm bus để kết nối Data Collector, AI Prediction, Backtesting và Sentiment Services.



Event Grid / Service Bus

nếu triển khai trên Azure/AWS, dùng dịch vụ native để tích hợp ứng dụng, cài đặt rules và routing.

Kết luận – Kiến trúc nên ưu tiên

Đối với đồ án hệ thống phân tích và dự báo tài chính AI:



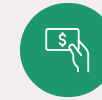
Nên chọn

Microservices + Containers + Message Broker + Event-Driven Architecture. Đây là bộ khung chủ đạo để xử lý dữ liệu thời gian thực, tách biệt chức năng, mở rộng linh hoạt và chịu lỗi.



Thiết kế

Áp dụng Domain-Driven Design và Clean Architecture để thiết kế các microservice có ranh giới rõ ràng và dễ bảo trì.



Giao dịch

Sử dụng Saga pattern cho giao dịch xuyên dịch vụ; áp dụng CQRS và Event Sourcing cho các phần yêu cầu phân tích và audit dữ liệu lịch sử.

Service Mesh

triển khai khi số lượng microservice lớn hoặc cần tính năng bảo mật và quan sát chi tiết; ban đầu có thể tạm bỏ để đơn giản hóa.

SPA Framework

thiết kế front-end thời gian thực với React/Vue, kết nối WebSocket.

Serverless / Lambda Architecture

dùng cho những tác vụ nhẹ, theo sự kiện (như gửi notification hoặc tái huấn luyện định kỳ). Lambda Architecture chỉ cần thiết nếu xử lý Big Data quy mô rất lớn; có thể đơn giản hóa với kiến trúc streaming thuần túy.

Software Integration

triển khai API Gateway, workflow orchestration và các dịch vụ tích hợp (Kafka, Service Bus) để kết nối toàn hệ thống.

Thiết kế này vừa đáp ứng yêu cầu học thuật (phân rã hệ thống, quản lý sự kiện, thực hiện AI prediction) vừa có tính thực tiễn cao.

Event-Driven Architecture (EDA)

- Hệ thống được thiết kế xoay quanh **sự kiện** – tức là một thông báo về việc gì đó vừa xảy ra.
- Một **sự kiện** chỉ là một thông điệp, ví dụ: *“Có tin tức mới”*, *“Giá BTCUSDT cập nhật”*, hoặc *“Phân tích cảm xúc hoàn tất”*.
- Các thành phần (microservice) **không gọi trực tiếp** nhau, mà:
 - a. **Producer** phát (publish) sự kiện.
 - b. **Broker** (như Kafka, RabbitMQ) nhận và phân phối.
 - c. **Consumer** lắng nghe (subscribe) và xử lý nếu quan tâm.

Cách này giúp dịch vụ **tách lỏng** (loose coupling): thêm, bớt, hoặc thay đổi dịch vụ mà không ảnh hưởng dịch vụ khác.

Event Streaming

- Là việc **thu thập**, **truyền** và **xử lý** các sự kiện **liên tục theo thời gian thực**.
- Hình dung như **dòng chảy dữ liệu** (stream) từ nguồn → nền tảng streaming → các bộ xử lý → nơi tiêu thụ.
- Hỗ trợ lưu trữ lâu dài và **phát lại** (replay) sự kiện khi cần (debug, huấn luyện AI...).

Luồng Market Data → AI Prediction → Charting

1. **Data Collector Service** publish giá BTCUSDT mỗi phút vào `marketdata.btcusdt.1m`.
2. **AI Prediction Service** subscribe stream này, tính feature, dự đoán → publish kết quả vào `prediction.output`.
3. **Real-time Charting Service** subscribe cả hai stream để vẽ overlay giá + dự đoán.