Problem 2: Building a Castle In The Cloud

A. Architect for the application

Please use this link and enter my code to see the diagram: https://mermaid.live

```
flowchart TB

%% --------------------------
%% AWS Organization (parent)
%% --------------------------
subgraph aws_org["AWS Organization (Multi-Account)"]

    %% ================================================
    %% Dev Account (Shortened)
    %% ================================================
    subgraph dev_acc["Dev Account"]
        direction TB
        subgraph dev_arch["Dev Core Architecture (Simplified)"]
            dev_edge["Edge (CF, WAF, ACM)"]
            dev_alb_api["ALB + API Gateway"]
            dev_eks["EKS/ECS + KEDA"]
            dev_services["Microservices & Service Mesh"]
            dev_db["Aurora/DynamoDB"]
            dev_logs["Logging (CloudWatch/X-Ray)"]
            dev_s3["S3 Data Lake (+Glue, etc.)"]
        end

        dev_edge --> dev_alb_api
        dev_alb_api --> dev_eks
        dev_eks --> dev_services
        dev_services --> dev_db
        dev_services --> dev_logs
        dev_s3 --> dev_logs
    end


    %% ================================================
    %% Test Account (Shortened)
```

```
%% ================================================
subgraph test_acc["Test Account"]
    direction TB
    subgraph test_arch["Test Core Architecture (Simplified)"]
        test_edge["Edge (CF, WAF, ACM)"]
        test_alb_api["ALB + API Gateway"]
        test_eks["EKS/ECS + KEDA"]
        test_services["Microservices & Service Mesh"]
        test_db["Aurora/DynamoDB"]
        test_logs["Logging (CloudWatch/X-Ray)"]
        test_s3["S3 Data Lake (+Glue, etc.)"]
    end

    test_edge --> test_alb_api
    test_alb_api --> test_eks
    test_eks --> test_services
    test_services --> test_db
    test_services --> test_logs
    test_s3 --> test_logs
end


%% ================================================
%% Prod Account (Full Detail)
%% ================================================
subgraph prod_acc["Prod Account"]
    direction TB

    %% -----------------
    %% Subgraph: Edge
    %% -----------------
    subgraph prod_edge["Edge Layer"]
        cf["Amazon CloudFront"]
        waf["AWS WAF"]
        acm["AWS Certificate Manager"]
        route53["Route 53 (DNS + Health Checks)"]
    end

    route53 --> cf
    cf --> waf
    waf --> acm

    %% -----------------
```

```
        %% Subgraph: API & Ingress
        %% -----------------
        subgraph prod_api["API & Ingress"]
            alb["Application Load Balancer"]
            apigw["Amazon API Gateway"]
            cognito["AWS Cognito / SSL Termination"]
        end

        acm --> alb
        alb --> apigw
        apigw --> cognito

        %% -----------------
        %% Subgraph: Microservices Orchestration
        %% -----------------
        subgraph prod_micro["Microservices Orchestration"]
            eks["EKS (or ECS) + KEDA"]
            svcMesh["Service Mesh (Istio/App Mesh)"]
            intApi["Internal API Gateway <br/> for Microservices"]
        end

        cognito --> eks
        eks --> svcMesh
        svcMesh --> intApi

        %% -----------------
        %% Subgraph: Core Trading Components
        %% -----------------
        subgraph prod_trading["Core Trading Components"]
            matchEng["Real-Time Matching Engine <br/>(Dedicated EC2 in
ASG)"]
            llm["Low Latency Messaging <br/>(Kinesis / Kafka)"]
        end

        eks --> llm
        llm --> matchEng

        %% -----------------
        %% Subgraph: Cache & Session
        %% -----------------
        subgraph prod_cache["Cache & Session Management"]
            redis["Amazon ElastiCache (Redis)"]
        end
```

```
intApi --> redis
matchEng --> redis

%% -----------------
%% Subgraph: Data & Transactional
%% -----------------
subgraph prod_data["Data & Transactional Storage"]
    aurora["Aurora / DynamoDB (Sharded/Global)"]
    acidDB["ACID-compliant Order DB"]
end

intApi --> aurora
matchEng --> aurora

%% -----------------
%% Subgraph: Monitoring & Security
%% -----------------
subgraph prod_sec["Security & Audit"]
    logs["CloudWatch, X-Ray"]
    enc["Encryption & IAM"]
end

eks --> logs
matchEng --> logs
aurora --> logs

%% -----------------
%% Subgraph: Historical & Analytics
%% -----------------
subgraph prod_analytics["Historical & Analytics"]
    s3["S3 w/ Lifecycle Policies"]
    glue["AWS Glue (Catalog/ETL)"]
    redshift["Amazon Redshift"]
    athena["Athena (Ad-hoc Queries)"]
    kda["Kinesis Data Analytics"]
end

s3 --> glue
s3 --> athena
glue --> redshift
kda --> s3
```

```
        %% Full Connections Inside Prod
        acm --> alb
        alb --> apigw
        apigw --> cognito
        cognito --> eks
        eks --> svcMesh
        svcMesh --> intApi
        intApi --> aurora
        intApi --> redis
        llm --> matchEng
        matchEng --> redis
        matchEng --> aurora
        aurora --> logs
        redis --> logs

        %% Additional flows to analytics
        intApi --> s3
        matchEng --> s3
        aurora --> s3
        s3 --> logs
    end

    %% Inter-Account Networking
    tgw["Transit Gateway or VPC Peering"]
    dev_acc --> tgw
    test_acc --> tgw
    prod_acc --> tgw
end

%% ------------------------------------
%% Security & Governance (Central)
%% ------------------------------------
subgraph sec_gov["Security & Governance"]
    secHub["AWS Security Hub"]
    guardDuty["Amazon GuardDuty"]
    awsConfig["AWS Config"]
    iamScp["IAM Roles & SCPs (Organizations)"]
end

aws_org --> sec_gov
```

1. Multi-Account Structure

In a mature AWS setup, you often separate environments (e.g., Dev, Test, Prod) into **different AWS accounts** to:

- **Isolate** resources so that issues or changes in Dev/Test do not affect Production.
- **Improve security** by limiting cross-environment access.
- **Simplify billing** (separate cost tracking for each environment).
- **Enforce compliance** using organizational policies.

Dev Account

- **Purpose**: Used for active development, experimentation, and integration testing. Developers have more freedom here to spin up services, test features, and iterate quickly.

Test Account

- **Purpose**: Dedicated for QA, staging, and performance testing. Ensures changes are validated (functional, performance, security) before pushing to production.

Prod Account

- **Purpose**: Hosts your live, customer-facing or mission-critical workloads. Must be highly secured, monitored, and follow strict change-management processes to ensure uptime and data integrity.

Transit Gateway / VPC Peering

- **Purpose**: Provides **network connectivity** between the separate AWS accounts.
- **Transit Gateway** scales better for multiple VPCs; **VPC Peering** is simpler for one-to-one connections.
- **Benefit**: Allows shared services or data replication between environments.

---

2. Edge Layer

Route 53 (DNS & Health Checks)

- **Purpose**: AWS' DNS service to direct user traffic to your application endpoints.
- **Key Capabilities**:
    - **Health Checks**: Automatically check endpoint health and reroute if one endpoint fails.
    - **Routing Policies**: Simple, Weighted, Geolocation, Latency-based, Failover, etc., to optimize domain traffic management.

Amazon CloudFront

- **Purpose**: Content Delivery Network (CDN) that caches and delivers content (static or dynamic) from edge locations close to users.
- **Benefit**: Improves **performance** (reduced latency) and **reduces load** on origins (e.g., ALB, S3).

AWS WAF (Web Application Firewall)

- **Purpose**: Protects web applications from common attacks (SQL injection, XSS, bots, etc.).
- **Integration**: Typically attaches to CloudFront or Application Load Balancers for centralized inspection of HTTP/S traffic.

AWS Certificate Manager (ACM)

- **Purpose**: Issues and manages **SSL/TLS certificates** for your domains.
- **Benefit**: Automates certificate **renewal**, integrates seamlessly with AWS services like **CloudFront**, **ALB**, and **API Gateway**.

---

3. API & Ingress

Application Load Balancer (ALB)

- **Purpose**: Distributes incoming HTTP(S) requests across multiple targets (EC2, ECS, EKS pods).
- **Key Features**: Path-based routing, host-based routing, WebSocket support, integration with AWS WAF.

Amazon API Gateway

- **Purpose**: Manages **REST, HTTP, or WebSocket** APIs. Offers features like **throttling**, **request validation**, and **usage plans**.
- **Benefit**: Simplifies exposing microservices as well-defined APIs; can integrate with **AWS Lambda**, **ECS/EKS**, or **VPC Link** to backend services.

AWS Cognito

- **Purpose**: Provides **user authentication**, **authorization**, and **user management** (signup/signin flows, secure tokens).
- **Integration**: Often used with API Gateway to handle **JWT-based** auth or federated SSO (e.g., Social logins, SAML).

---

4. Microservices Orchestration

Amazon EKS (or ECS) + KEDA

- **Purpose**:
    - **EKS**: Managed Kubernetes control plane, schedules containerized apps (pods) across EC2 or Fargate.
    - **ECS**: Alternative AWS container service with simpler operational overhead.
    - **KEDA** (Kubernetes-based Event Driven Autoscaling): Monitors external event sources (e.g., SQS, Kinesis) and **auto-scales** pods based on queue depth or event volume.
- **Benefit**: Scalability, flexibility, standardized container deployment.

Service Mesh (Istio / App Mesh)

- **Purpose**: Controls **service-to-service communication** with features like **traffic routing**, **mTLS encryption**, **observability** (metrics, logs, tracing), and **security policies**.
- **Benefit**: Offloads complex networking logic from individual microservices to a **sidecar proxy**.

Internal API Gateway

- **Purpose**: Some organizations prefer an **internal** gateway for microservices to unify internal API routes, apply consistent policies, or handle zero-trust networking behind the service mesh.
- **Note**: Depending on complexity, the service mesh alone might handle these responsibilities without a separate gateway.

---

5. Core Trading Components (Example: High-performance, finance-related)

Real-Time Matching Engine (EC2 in ASG)

- **Purpose**: Processes **trade orders** and matches them in real-time with minimal latency.
- **Why EC2**: If you need **low-level OS access**, specialized libraries, or consistent ultra-low latency, a dedicated EC2 cluster may outperform containers.
- **Auto Scaling Group** (ASG): Automatically adds or removes EC2 instances based on CPU, memory, or custom metrics (e.g., orders/sec).

Low Latency Messaging (Kinesis / Kafka)

- **Purpose**: Ingestion and distribution of **streaming data** (trades, events) at scale.

- **Kinesis** is a fully managed AWS service; **Apache Kafka** (or MSK) offers a popular open-source alternative for advanced streaming.
- **Benefit**: Real-time data pipelines, decoupled microservices, backpressure handling, replay capabilities.

---

6. Cache & Session Management

Amazon ElastiCache (Redis)

- **Purpose**: High-performance **in-memory** caching layer for session data, frequently accessed queries, or real-time analytics.
- **Benefit**: Reduces latency, offloads reads from the database, can manage user sessions at scale.

---

7. Data & Transactional Storage

Aurora / DynamoDB (Sharded/Global)

- **Aurora** (Relational): MySQL/PostgreSQL-compatible. Offers **ACID** transactions, read replicas, and global database features for **low-latency cross-region** reads and rapid failover.
- **DynamoDB** (NoSQL): Scales horizontally for key-value or document store workloads, supports global tables for multi-region replication, and provides **eventual** or **strong** consistency.
- **Sharding**: A strategy to split large volumes of data across multiple DB instances or tables.

ACID-Compliant Order DB

- **Purpose**: If you need strict, **atomic** transaction guarantees (e.g., for financial trades), an **ACID** database ensures data consistency even in high-throughput scenarios.
- **Benefit**: Preserves transactional integrity (e.g., no partial trades in the system).

---

8. Security & Audit

CloudWatch, X-Ray

- **Purpose**: Centralized **logging**, **metrics**, and **tracing** for distributed systems.
- **CloudWatch**: Collects logs (e.g., from ECS tasks, EKS pods), infrastructure metrics, and custom application metrics.

- **X-Ray**: Enables **distributed tracing** for microservices to diagnose performance bottlenecks and trace requests end-to-end.

Encryption & IAM

- **Purpose**:
  - **Encryption**: Secure data at rest (e.g., in EBS, RDS, S3) and in transit (TLS).
  - **IAM**: Access control for users, services, and resources with **least-privilege** policies.
- **Benefit**: Helps meet compliance (HIPAA, PCI-DSS, etc.) and ensures only authorized actors can access data.

---

9. Historical & Analytics

Amazon S3 (w/ Lifecycle Policies)

- **Purpose**: Durable, scalable object storage for raw data, logs, archives, backups.
- **Lifecycle Policies**: Automatically transition older data to cheaper classes (e.g., Glacier).

AWS Glue (Catalog/ETL)

- **Purpose**: Data **catalog** and ETL service. Glue **crawlers** detect schema in S3 data, making it queryable in Athena or Redshift Spectrum.
- **Benefit**: Simplifies building a **data lake** in S3.

Amazon Redshift

- **Purpose**: Fully managed, **petabyte-scale** data warehouse for analytics and BI dashboards.
- **Integration**: Can query data in S3 via **Redshift Spectrum**.

Amazon Athena

- **Purpose**: Serverless, interactive query service for data in S3.
- **Benefit**: Great for ad-hoc queries, does not require managing any infrastructure.

Kinesis Data Analytics

- **Purpose**: Real-time analytics on streaming data (from Kinesis or Kafka).
- **Benefit**: Perform **windowed** operations, real-time metrics, or anomaly detection without building a complex pipeline.

---

10. Security & Governance (Organization-wide)

AWS Security Hub

- **Purpose**: A unified dashboard that aggregates security findings from **GuardDuty**, **Inspector**, **Macie**, and other tools. Provides a **single pane of glass** for compliance checks (CIS, PCI, etc.).

Amazon GuardDuty

- **Purpose**: Continuous **threat detection** service that uses machine learning to identify malicious activity or unauthorized behavior.
- **Monitors** CloudTrail, VPC Flow Logs, DNS logs, etc.

AWS Config

- **Purpose**: Tracks **configuration changes** of AWS resources. Allows you to define rules (e.g., “S3 buckets must not be public”) to enforce compliance.
- **Benefit**: Provides a historical record of resource config changes for audits.

IAM Roles & Service Control Policies (SCPs)

1. **Purpose**:
    a. **IAM Roles**: Grants permissions to AWS services or users based on the principle of **least privilege**.
    b. **SCPs**: Organizational-level “guardrails” that limit what actions are allowed across **all** accounts or within specific Organizational Units (OUs).
2. **Benefit**: Prevents accidental or malicious usage of restricted services/actions across Dev/Test/Prod.

B. The 5 main features that we can cover with this architect

1. Real-Time Order Matching and Execution

Key Services

- **Real-Time Matching Engine (EC2 in ASG)**
- **Low-Latency Messaging (Kinesis/Kafka)**
- **Microservices Orchestration (EKS/ECS + Service Mesh)**
- **ElastiCache (Redis)**

Explanation

1. **Dedicated Matching Engine**:
    - Placed on **EC2 instances in an Auto Scaling Group**, allowing you to run specialized low-latency code (e.g., C++/Rust) without container overhead.
    - You can fine-tune OS settings, CPU pinning, and networking (ENA drivers) to minimize jitter—essential for high-frequency trading.
2. **Low-Latency Messaging**:
    - **Kinesis** or **Kafka** ensures **real-time ingestion** of new orders.
    - Messaging decouples order placement from the matching engine, preventing direct coupling that can lead to bottlenecks.
    - **Partitioning** or **shards** can align with specific trading pairs or symbol sets, keeping throughput high and latency predictable.
3. **Microservices & Orchestration**:
    - **EKS/ECS** handles ancillary microservices (e.g., user profile, risk checks, trade history) that interact with the matching engine via APIs or messaging queues.
    - A **Service Mesh** like Istio or AWS App Mesh ensures consistent, **secure** service-to-service communication with built-in **observability** (metrics/traces).
4. **In-Memory Cache (Redis)**:
    - Used for **hot data** (e.g., open orders, rapidly updated user balances).
    - Reduces round trips to the database, keeping latencies low for real-time lookups.

Result: By decoupling order ingestion (Kinesis/Kafka) from order matching (EC2/ASG) and caching (Redis), you get **very fast, real-time** execution with minimal overhead.

---

2. Transaction Integrity and ACID Compliance

Key Services

- **Amazon Aurora (SQL) / DynamoDB (NoSQL with Transactions)**
- **ACID-Compliant Order & Transaction Database**
- **Microservices Orchestration**

Explanation

1. **ACID-Compliant Database**:
    - **Aurora** (MySQL/Postgres-compatible) provides **strong ACID guarantees** with built-in replication, point-in-time recovery, and multi-AZ durability.
    - If using **DynamoDB**, transactions at the item-level can **atomically** update multiple items, ensuring partial updates do not occur.
2. **Atomic Operations**:
    - In a trading context, partial trades or inconsistent states can cause major discrepancies.
    - Aurora's **row-level locking** or DynamoDB's **transaction support** helps maintain data integrity (e.g., account balance updates, order status changes).
3. **Microservices Boundaries**:
    - Each microservice (e.g., "Order Service," "Portfolio Service," etc.) interacts with a transactional store.
    - Patterns like **sagas** or **two-phase commit** (if needed) coordinate multi-service transactions—though often you keep each transaction bounded in one domain service to keep it simpler.
4. **Event Sourcing & Auditing**:
    - Data changes can be **streamed** to Kinesis/Firehose for an immutable record of all trades.
    - This also helps with compliance/regulatory audits.

Result: Ensuring each trade or balance update is handled in a **transactional manner** prevents double-spend errors and ensures **full data consistency** for critical financial operations.

---

3. Handling High Traffic and Peak Loads

Key Services

- **Autoscaling (EKS/ECS with KEDA / EC2 ASG)**
- **Edge (CloudFront, WAF, ACM) + ALB/API Gateway**
- **ElastiCache**
- **Kinesis/Kafka (Partitioned Streams)**

Explanation

1. **Autoscaling Everywhere**:

    - **EKS/ECS** can horizontally scale microservices based on CPU, memory, or custom metrics (via **KEDA**).
    - **EC2 Auto Scaling Groups** add or remove matching engine nodes under heavy workloads (e.g., major market movements).
    - **Kinesis** or **Kafka** partition scaling ensures message throughput can handle high volumes of orders.
2. **Edge Layer**:
    - **CloudFront** caches static and semi-static content (e.g., the trading dashboard's static assets), reducing load on application backends.
    - **AWS WAF** filters malicious requests early, preventing resource waste from potential DDoS attempts or bot traffic.
3. **Caching**:
    - **Redis** for frequently accessed data (e.g., market data, real-time order book snapshots).
    - Minimizes repeated queries on primary databases, thus **reducing latency** and improving throughput under surge traffic.
4. **API Gateway + ALB**:
    - Splits traffic: high-throughput, lower-level requests might route through an **ALB** directly, while advanced API-based flows go through **API Gateway**.
    - **Usage plans** and **throttling** help prevent service overload.

Result: With robust **scaling** at every layer and effective **caching/streaming**, the system can handle massive peaks—akin to what major exchanges face during volatile crypto market events.

---

4. Advanced Security and DDoS Mitigation

Key Services

- **AWS WAF + CloudFront**
- **AWS Shield (Standard or Advanced)**
- **IAM Roles & SCPs**
- **Security Hub, GuardDuty, Config**
- **Encryption (ACM, TLS, KMS)**

Explanation

1. **WAF + CloudFront**:
    - **WAF** can block malicious patterns (SQL injection, XSS) at the edge before they reach your application.
    - **CloudFront** offloads TLS termination, absorbs spikes, and integrates with **AWS Shield** to mitigate DDoS at the network layer (L3/L4).
2. **Advanced Threat Detection**:
    - **GuardDuty** continuously analyzes VPC flow logs, CloudTrail logs, and DNS queries for suspicious activities (e.g., port scans, anomalous traffic).

    - Findings feed into **Security Hub**, giving a unified security dashboard for all AWS accounts.
3. **IAM & SCPs**:
    - **IAM** roles with least-privilege policies ensure microservices and DevOps engineers have only the necessary access.
    - **Service Control Policies (SCPs)** restrict high-risk actions or services across Dev/Test/Prod at the organizational level.
4. **Encryption**:
    - Data at rest in **Aurora**, **DynamoDB**, **S3**, and **ElastiCache** can be encrypted with AWS KMS.
    - In-transit encryption (TLS/SSL) enforced at CloudFront, ALB, or API Gateway using **AWS Certificate Manager**.
    - **mTLS** inside the service mesh can protect internal service-to-service traffic from snooping.
5. **DDoS Mitigation**:
    - **AWS Shield Standard** automatically included with CloudFront and Route 53 to handle common DDoS attacks.
    - **AWS Shield Advanced** (optional) provides more sophisticated detection and cost protection during large-scale attacks.

Result: An advanced multi-layer security approach from edge to data ensures the platform remains **secure and compliant**, even under targeted DDoS or other sophisticated threats.

---

5. Crypto-Specific Features and Compliance

Key Services

- **Multi-Account Setup** (Dev/Test/Prod)
- **Audit Logging (CloudWatch, X-Ray, S3)**
- **IAM, Config, KMS**
- **Data Lake** (S3, Glue, Redshift) for analytics and forensics
- **(Optional) HSM / AWS CloudHSM** for private key management if needed

Explanation

1. **Regulatory Compliance & Audits**:
    - Many crypto exchanges must adhere to KYC/AML regulations, **FATF guidelines**, or local financial authorities.
    - You can keep user data, logs, and transaction records in **separate accounts** (lower blast radius), track changes with **Config**, and gather logs in **CloudWatch** or **S3** for audit trails.
2. **Data Lake for Transaction & Trade History**:
    - Storing all trades and user events in **S3** (with **Glue Data Catalog**) provides an **immutable** record of historical data for compliance and forensic analysis.
    - **Encryption at rest** ensures customer data confidentiality.

    - **Lifecycle Policies** automatically transition older data to cheaper storage (Glacier), meeting **long-term** record-keeping requirements.
3. **Secure Key Management**:
    - **AWS KMS** can manage encryption keys for data at rest.
    - If you need hardware-level cryptographic modules for cold or warm wallets, **AWS CloudHSM** or external HSMs can integrate with your exchange.
    - This is critical for **custody of crypto assets**, though many large exchanges use separate hardware wallets or third-party custody solutions.
4. **Isolated Dev/Test/Prod**:
    - Ensures **production** cryptographic keys, wallets, and user data remain **completely separate** from development environments, reducing the risk of internal fraud or accidental data leaks.
5. **Integration with Identity Providers**:
    - **Cognito** can integrate with user pool logic for KYC checks or 2FA (MFA), while additional checks or AML logic can be implemented at the application layer or via 3rd-party APIs.

Result: By leveraging **multi-account isolation**, **secure data storage**, **robust logging**, and optional **CloudHSM** for key management, your architecture meets the **unique compliance and security needs** of a crypto trading platform.