Architecture Documentation – Online Store & Mahak Integration

1. Overview

This document provides a formal architectural description of the integration between the Online Store System (Hybrid architecture) and the Mahak Accounting System. This integration is crucial for maintaining synchronized financial data, enabling accurate invoicing, payment tracking, and overall financial health of the online store operations. The document outlines the system's context, its major architectural components, detailed breakdowns of key modules, the chosen technology stack with thorough justifications, and illustrative sequence diagrams to depict crucial synchronization processes. The architecture is designed to be scalable, resilient, and maintainable, leveraging modern cloud-native principles where applicable.

---

2. System Context (C1)

The System Context diagram (C1) visualizes the external entities and systems that interact with the Online Store System, highlighting the boundaries and interfaces of our solution. It provides a high-level understanding of how the Online Store fits within its broader operational ecosystem.

Primary Actors:

- Customer:
    - Description: The end-user who interacts with the online store to purchase products.
    - Interactions:
        - Registration & Authentication: Creates an account and logs in securely.
        - Browsing: Views product catalogs, details, and pricing.

        - Ordering: Adds items to a cart, proceeds to checkout, and places orders.
        - Payment: Initiates and completes payment transactions.
        - Notifications: Receives order confirmations, shipping updates, and other transactional messages.

- Admin:
    - Description: The internal user responsible for managing the online store's operations.
    - Interactions:
        - Product Management: Adds, updates, and removes products, manages inventory.
        - Order Management: Views, processes, and fulfills customer orders.
        - Content Management: Manages website content, promotions, and user accounts.
        - Reporting: Accesses sales and operational reports.

- Payment Gateway:
    - Description: An external financial service provider that securely processes customer payments.
    - Interactions:
        - Payment Processing: Receives payment requests from the Online Store and securely handles transaction authorization and settlement.
        - Payment Confirmation: Sends back transaction status (success/failure) to the Online Store.

- SMS Provider:
    - Description: An external service used for sending time-sensitive notifications to customers and potentially administrators.
    - Interactions:
        - OTP Sending: Delivers One-Time Passwords for user authentication and verification.
        - Transactional Notifications: Sends critical updates like order confirmations, shipping alerts, and payment receipts.

- Mahak Accounting System:
    - Description: The core financial and accounting system that manages the business's financial records. The integration facilitates seamless data exchange.
    - Interactions:
        - Invoice Exchange: Sends sales invoices generated by the online store to Mahak for accounting. Receives accounting status updates or credit memos from Mahak.
        - Payment Reconciliation: Sends payment confirmations to Mahak for reconciliation with outstanding invoices. Receives payment status or account balance information.
        - Account Status Updates: Exchanges relevant customer or account status information as needed.

Key Interactions Summary:

- Customer ↔ Online Store: The primary user interface for all customer-facing operations, from account creation to order fulfillment.
- Admin ↔ Online Store: The administrative interface for managing all backend operations of the online store.
- Online Store ↔ Payment Gateway: Critical for secure and reliable transaction processing. This is a synchronous, real-time interaction.
- Online Store ↔ SMS Provider: Used for time-sensitive communication, primarily for security and transactional updates. This is typically a synchronous, API-based interaction.
- Online Store ↔ Mahak API: This is the core integration point for financial data synchronization. It involves asynchronous data exchange for invoices, payments, and other financial records. This interaction is typically managed via a dedicated integration worker and a message queue to ensure reliability and decouple systems.

---

3. Containers (C2)

The Containers diagram (C2) breaks down the Online Store System and its immediate infrastructure into deployable units, illustrating the major subsystems and their interactions.

Containers:

1. API Gateway / BFF (Backend For Frontend):
    - Description: A single entry point for all client applications (Frontend, potential mobile apps). It handles cross-cutting concerns like authentication, authorization, rate limiting, logging, and request aggregation. A BFF pattern allows tailoring API responses to the specific needs of different frontends.
    - Technology: Typically built using .NET Core/ASP.NET Core Web API, Go, or Node.js, often deployed behind an API Gateway solution like Ocelot, Kong, or managed cloud gateway services.
2. Local Database (PostgreSQL 15):
    - Description: A relational database instance managed by the Online Store system. It stores essential operational data such as product information, customer profiles, orders, and critically, a replicated or transformed subset of data synced from Mahak for operational efficiency and auditing. PostgreSQL's `JSONB` support is leveraged for flexible data handling.
    - Technology: PostgreSQL 15.
3. Message Queue (RabbitMQ):
    - Description: A robust message broker that facilitates asynchronous communication between different parts of the system, especially for integration tasks. It ensures reliable message delivery, decoupling of producers and consumers, and supports patterns like publish-subscribe and worker queues.
    - Technology: RabbitMQ.
4. Mahak Sync Worker:
    - Description: A dedicated background service responsible for orchestrating the data synchronization process with the Mahak Accounting System. It pulls data from Mahak via its API, transforms it, and pushes it to the Message Queue for further processing and persistence in the Local Database. This container is designed for periodic execution or event-driven triggering.

- Technology: Built using .NET 8 or Go, running as a standalone executable or containerized application.

5. Object Storage (MinIO/S3):
    - Description: A scalable storage solution for unstructured data, primarily used for storing product images, downloadable assets, and potentially backup files. MinIO is chosen for its S3-compatible API, allowing for both cloud-based (AWS S3) and on-premise deployments.
    - Technology: MinIO (for on-premise/private cloud) or AWS S3 (for cloud).
6. Mahak API Adapter:
    - Description: A conceptual layer or a dedicated service that encapsulates all interactions with the Mahak Accounting System's API. It handles authentication with Mahak, request formatting, response parsing, error handling, and provides a consistent interface for the Mahak Sync Worker. This adapter shields the sync worker from the intricacies of the Mahak API.
    - Technology: Implemented within the Mahak Sync Worker or as a separate microservice, using .NET 8 or Go, with HTTP client libraries.
7. Frontend:
    - Description: The user interface layer for both customers and administrators. This is typically a Single Page Application (SPA) or a server-side rendered application that consumes APIs exposed by the API Gateway.
    - Technology: React, Angular, Vue.js (for SPAs) or ASP.NET Core MVC/ Razor Pages (for server-side rendering).

---

4. Components – Mahak Sync Worker (C3)

The Mahak Sync Worker is a critical component responsible for the bidirectional or unidirectional synchronization of data between the Online Store and the Mahak Accounting System. This section details its internal modules and their functions.

Key Modules:

1. RowVersion Tracker:
    - Purpose: To implement incremental synchronization. Instead of fetching all data every time, this module tracks the last processed `RowVersion` (or a similar timestamp/identifier) for each entity from Mahak. Subsequent requests to Mahak's API will only fetch records that have a `RowVersion` greater than the last processed one, significantly reducing data transfer and processing load.
    - Functionality:
        - Stores the latest `RowVersion` successfully processed for each entity type (e.g., Invoices, Payments).
        - Retrieves this `RowVersion` before initiating a sync job.
        - Updates the stored `RowVersion` after a successful batch of records is processed.
        - Handles cases where Mahak's API might return records with the same `RowVersion` to ensure no data is missed.
2. Bulk Fetcher/Saver (Paging):
    - Purpose: To efficiently retrieve large volumes of data from Mahak's API and to insert/update this data into the local PostgreSQL database in manageable chunks (batches).
    - Functionality:
        - Paging: Implements logic to make multiple API calls to Mahak, requesting data in pages (e.g., `PageSize`, `Offset` or cursor-based pagination) to avoid overwhelming the API or the worker.
        - Batching: Collects records from Mahak's paginated responses and groups them into batches suitable for database operations (e.g., 100-1000 records per batch).
        - Database Operations: Executes bulk `INSERT` or `UPDATE` statements against the PostgreSQL database for efficiency.
3. Upsert Engine:
    - Purpose: To handle the insertion or update of records into the local database idempotently. This means that if the same data is

processed multiple times, the end state of the database remains consistent and correct without creating duplicate entries or causing errors.

- Functionality:
  - For each incoming record, it checks if a record with the same unique identifier (e.g., Mahak's Invoice ID) already exists in the local database.
  - If it exists, it performs an `UPDATE`.
  - If it does not exist, it performs an `INSERT`.
  - Utilizes database constraints (e.g., `UNIQUE` constraints on Mahak IDs) and `ON CONFLICT` clauses (in PostgreSQL) for efficient upsert operations.

4. Logical Delete Handler:

- Purpose: To manage the synchronization of deleted records. Instead of physically deleting records from the local database, a "soft delete" mechanism is preferred to maintain historical integrity and traceability.
- Functionality:
  - Detects records marked as deleted in Mahak's API response.
  - Marks corresponding records in the local database as "deleted" (e.g., by setting a boolean `IsDeleted` flag or a `DeletedAt` timestamp) rather than removing them entirely.
  - Ensures that future fetches from Mahak do not re-introduce logically deleted records if they are not explicitly reactivated.

5. Retry & Error Manager:

- Purpose: To ensure the robustness and reliability of the synchronization process by handling transient failures and persistent errors gracefully.
- Functionality:
  - Retry Logic: Implements exponential backoff strategies for transient API errors (e.g., network issues, temporary service unavailability) or database errors.
  - Dead Letter Queue (DLQ): Unprocessable messages or records that repeatedly fail after multiple retries are moved to a

dedicated DLQ (in RabbitMQ or a separate table in the DB). This prevents blocking the entire processing pipeline.
        - Error Reporting: Logs detailed error information, including the failed record, error message, and stack trace. Integrates with monitoring and alerting systems (e.g., Prometheus, Grafana, Sentry) to notify operations teams of critical failures.
        - Idempotency Checks: Reinforces the Upsert Engine's role to ensure that even if a message is processed multiple times due to retries, the outcome is consistent.

6. Data Transformer:

    - Purpose: To map data payloads received from the Mahak API to the specific schema and format required by the Online Store's Local Database. This is crucial because external API schemas rarely match internal database schemas directly.
    - Functionality:
        - Parses JSON or XML responses from Mahak.
        - Performs data type conversions (e.g., string to date, string to number).
        - Renames fields to match local database column names.
        - Handles missing or optional fields, providing default values where appropriate.
        - Performs any necessary data enrichment or calculations before saving to the local database.
        - Maps Mahak's status codes or identifiers to internal representations.

---

5. Code / Implementation Details (C4)

The Code/Implementation Details diagram (C4) provides a more granular view of the technologies and patterns used within the system's components, focusing on the core backend services.

Technology Stack:

- Backend Services & API Implementations (.NET 8 / Go):
  - Description: The choice between .NET 8 and Go depends on team expertise, performance requirements, and existing infrastructure. Both are modern, high-performance, compiled languages suitable for building robust backend services and APIs.
    - .NET 8: Offers a mature ecosystem, excellent tooling, a rich set of libraries, and strong support for building web APIs (ASP.NET Core), background services, and microservices. It's particularly well-suited for applications requiring complex business logic and integrations.
    - Go (Golang): Known for its simplicity, concurrency features, and excellent performance, making it a strong candidate for microservices, high-throughput APIs, and system-level utilities like workers.
  - Justification: High performance, modern language features, strong community support, cross-platform compatibility.
- Local Database (PostgreSQL 15):
  - Description: A powerful, open-source relational database system. It supports advanced features like JSONB for storing semi-structured data, robust indexing (including GIN for JSONB), and ACID compliance, making it ideal for transactional data and integration data.
  - Justification: Cost-effectiveness, strong performance, excellent Linux compatibility, ease of containerization and orchestration, advanced features like JSONB and GIN indexes are highly beneficial for handling varied API data and complex queries.
- Message Queue (RabbitMQ):
  - Description: A widely adopted, feature-rich message broker that implements the Advanced Message Queuing Protocol (AMQP). It provides reliable message delivery, message persistence, routing capabilities (exchanges and queues), and support for various messaging patterns.

    - Justification: Proven reliability, flexibility in routing and message patterns, message persistence guarantees, dead-lettering capabilities for robust error handling, and mature management tools.

- JWT Authentication:
    - Description: JSON Web Tokens are used for secure, stateless authentication. Upon successful login, the system issues a JWT to the client. This token contains user claims and is signed by the server. Subsequent requests include the JWT in the `Authorization` header, which the API Gateway or individual services validate to authenticate the user.
    - Justification: Stateless nature allows for horizontal scaling of services without shared session state. JWTs are compact, URL-safe, and can be digitally signed, ensuring integrity and authenticity.

- Object Storage (MinIO/S3):
    - Description: A distributed, highly available object storage system compatible with the Amazon S3 API. It can be deployed on-premises or in a private cloud environment, offering a scalable solution for storing large binary objects like images and files.
    - Justification: Scalability, durability, S3 compatibility allows for easy integration with existing tools and potential migration to cloud object storage services. Provides a cost-effective solution for storing static assets.

- Kubernetes:
    - Description: An open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It manages the lifecycle of containers, ensuring high availability, self-healing, and efficient resource utilization.
    - Justification: Provides robust capabilities for:
        - Scalability: Automatically scales applications based on demand.
        - High Availability: Ensures applications remain available through mechanisms like replication and automatic restarts.
        - Self-Healing: Restarts failed containers, replaces unhealthy ones, and reschedules them onto healthy nodes.

- Zero-Downtime Deployments: Enables rolling updates and rollbacks without service interruption.
- Resource Management: Efficiently utilizes underlying compute resources.

---

6. Sequence Diagram — BulkSync (Simple)

Purpose: This simplified sequence diagram provides a high-level overview of the bulk synchronization process between the Online Store and Mahak, designed for non-technical stakeholders to grasp the overall flow of data.

```
sequenceDiagram
    actor Scheduler
    participant MahakAPI as Mahak API
    participant MahakSyncWorker as Mahak Sync Worker
    participant MessageQueue as Message Queue (RabbitMQ)
    participant LocalDatabase as Local DB (PostgreSQL)

    Scheduler->>MahakSyncWorker: Trigger Bulk Sync Job
    MahakSyncWorker->>MahakAPI: Request Data (e.g., Invoices)
    MahakAPI-->>MahakSyncWorker: Return Paginated Data
    loop Process Batches
        MahakSyncWorker->>MessageQueue: Publish Transformed Data Batch
        MessageQueue->>LocalDatabase: Consume & Save Data
        Note over LocalDatabase: Upsert records
    end
```

Flow:

1. Scheduler triggers Mahak Sync Worker: A scheduler (e.g., Cron, Kubernetes CronJob) initiates the Mahak Sync Worker process.
2. Mahak API returns data: The Mahak Sync Worker makes a request to the Mahak API to fetch relevant data (e.g., invoices, payments) that needs to be synchronized. The API returns data in a paginated format.
3. Data pushed to message queue: The Mahak Sync Worker processes the fetched data, transforms it into the appropriate format, and pushes it as individual messages or batches onto the RabbitMQ message queue.

4. Local DB updated: Dedicated consumers (part of the worker or separate services) read messages from the queue, perform necessary upsert operations, and save the data into the Local PostgreSQL database.

---

7. Sequence Diagram — BulkSync (Detailed)

Purpose: This detailed sequence diagram illustrates the technical flow of the bulk synchronization process, highlighting error handling, retries, and message queue interactions.

```
sequenceDiagram
    actor Scheduler
    participant MahakAPIAdapter as Mahak API Adapter
    participant MahakSyncWorker as Mahak Sync Worker
    participant RabbitMQ as Message Queue (RabbitMQ)
    participant LocalDatabase as Local DB (PostgreSQL)

    Scheduler->>MahakSyncWorker: Initiate Sync Task (Entity: Invoice)
    MahakSyncWorker->>MahakSyncWorker: Load Last Processed RowVersion
    MahakSyncWorker->>MahakAPIAdapter: Get Invoices (from RowVersion=X,
    MahakAPIAdapter->>MahakAPIAdapter: Authenticate with Mahak API
    MahakAPIAdapter->>MahakAPI: GET /api/v1/invoices?rowVersion=X&page=
    MahakAPI-->>MahakAPIAdapter: 200 OK { InvoiceData: [...], NextPageT
    MahakAPIAdapter-->>MahakSyncWorker: Return InvoiceData List
    alt Data available
        MahakSyncWorker->>MahakSyncWorker: Transform InvoiceData to Int
        MahakSyncWorker->>MahakSyncWorker: Prepare Batch for DB (e.g.,
        loop For each Batch
            MahakSyncWorker->>RabbitMQ: Publish InvoiceBatch Message (t
            RabbitMQ->RabbitMQ: Route message to appropriate queue
        end
        MahakSyncWorker->>MahakSyncWorker: Update Last Processed RowVer
    end
    opt If more pages exist
        MahakSyncWorker->>MahakAPIAdapter: Get Invoices (from RowVersio
        ... (repeat data fetching and publishing) ...
```

```
    end

    par RabbitMQ Consumer Processing
        participant SyncConsumer as Sync Consumer
        RabbitMQ->>SyncConsumer: Deliver InvoiceBatch Message
        SyncConsumer->>SyncConsumer: Validate Message Payload
        SyncConsumer->>LocalDatabase: Execute Bulk Upsert (INSERT/UPDAT
        alt DB Write Success
            SyncConsumer->>RabbitMQ: ACK Message
            Note over SyncConsumer: Log successful batch processing
        else DB Write Failure (e.g., constraint violation, transient er
            SyncConsumer->>RabbitMQ: NACK Message (Requeue or send to D
            Note over SyncConsumer: Log DB write error
            opt Retry Mechanism
                SyncConsumer->>RabbitMQ: Requeue Message (with delay)
            else Max Retries Reached
                SyncConsumer->>RabbitMQ: Send to Dead Letter Queue (DLQ
                Note over RabbitMQ: Stores unrecoverable messages
            end
        end
        opt Handle API Rate Limits/Timeouts (within adapter/worker)
            Note over MahakAPIAdapter: Implement circuit breaker, backo
        end
    end
```

Flow:

1. Scheduler triggers Sync Task: The scheduler initiates the sync process for a specific entity (e.g., Invoices).
2. Load Last Processed RowVersion: The worker retrieves the last successfully processed `RowVersion` for the entity to enable incremental fetching.
3. Get Data from Mahak API: The worker, via the Mahak API Adapter, makes a call to the Mahak API, requesting data starting from the last `RowVersion` and utilizing pagination. The adapter handles authentication and potentially initial error checking before forwarding the request.

4. Mahak API Returns Data: The Mahak API responds with a batch of data, possibly including a token or indicator for the next page.
5. Transform and Batch Data: The worker transforms the raw API data into the internal domain model and prepares it into batches suitable for efficient database insertion/updates.
6. Publish to RabbitMQ: Each data batch is published as a message to a designated exchange in RabbitMQ. The exchange routes it to a specific queue for invoice synchronization.
7. Update Last Processed RowVersion: After successfully preparing data for all pages, the worker updates the stored `RowVersion` for the next sync cycle.
8. RabbitMQ Consumer Processing:
    - A dedicated consumer instance picks up messages from the RabbitMQ queue.
    - It validates the message payload and executes a bulk `UPSERT` operation against the Local PostgreSQL database.
    - On Success: The consumer acknowledges the message (`ACK`), signaling successful processing.
    - On Failure: If the database write fails (e.g., due to transient network issues, constraint violations, or data integrity problems), the consumer negatively acknowledges the message (`NACK`).
        - Requeue: The message can be requeued for later processing, often with a delay mechanism.
        - Dead Letter Queue (DLQ): If a message consistently fails after a configured number of retries, it's sent to a DLQ. This prevents it from blocking other messages and allows for manual investigation of problematic data.
9. API Rate Limits/Timeouts: The Mahak API Adapter includes logic to manage API rate limits and handle timeouts by implementing retry mechanisms with exponential backoff.

---

8. Technology Choice Justifications

This section elaborates on the reasoning behind selecting specific technologies for the integration architecture.

- PostgreSQL over SQL Server:
  - Cost-Effectiveness: PostgreSQL is open-source, significantly reducing licensing costs compared to SQL Server, especially for large deployments.
  - Linux Compatibility: PostgreSQL has first-class support and seamless integration on Linux, which is the dominant operating system in cloud and containerized environments. SQL Server's Linux support, while improved, is still more mature on Windows.
  - Kubernetes Friendliness: PostgreSQL has excellent community support and tooling for running within Kubernetes clusters (e.g., Operators, Helm charts), making deployment and management straightforward.
  - JSONB and Indexing Capabilities: PostgreSQL's `JSONB` data type provides efficient storage and querying of semi-structured data, which is invaluable when integrating with APIs that may have evolving or flexible schemas. Advanced indexing capabilities (GIN, GiST) further enhance query performance for JSONB content.
  - Open Standards: Adherence to SQL standards and open-source principles promotes flexibility and reduces vendor lock-in.
- RabbitMQ:
  - Reliability and Durability: RabbitMQ offers robust message persistence, ensuring that messages are not lost even in case of broker restarts. It supports various acknowledgement modes for guaranteed delivery.
  - Flexibility and Routing: Its powerful exchange/queue routing mechanism allows for flexible message distribution patterns (e.g., direct, topic, fanout), which are essential for complex integration scenarios.

    - Mature Ecosystem: RabbitMQ is a battle-tested and widely adopted message broker with extensive documentation, community support, and tooling for management and monitoring.
    - Dead-Lettering: Built-in support for Dead Letter Queues is critical for handling processing failures gracefully, isolating problematic messages, and preventing pipeline blockage.

- MinIO/S3:

    - Scalability: Designed to handle massive amounts of data and scale horizontally to meet growing storage needs.
    - Durability and Availability: Object storage systems typically offer high durability through data replication and provide high availability for accessing stored objects.
    - S3 Compatibility: MinIO's adherence to the S3 API makes it interoperable with a vast ecosystem of tools and services, facilitating cloud migration and hybrid cloud strategies.
    - Cost-Effectiveness: For storing large binary assets like product images, object storage is often more cost-effective than block storage or relational databases.

- .NET 8:

    - High Performance: .NET Core and subsequent versions have undergone significant performance optimizations, making it a competitive choice for building high-throughput backend services.
    - Modern Features: Provides a rich set of modern language features, asynchronous programming capabilities, and a comprehensive framework (ASP.NET Core) for building APIs and web applications.
    - Developer Productivity: Strong tooling (Visual Studio, VS Code), extensive libraries, and a well-defined ecosystem contribute to faster development cycles and increased developer productivity.
    - Cross-Platform: Fully cross-platform, allowing seamless development and deployment on Windows, Linux, and macOS, which aligns well with containerization strategies.

- Kubernetes:
    - Container Orchestration Standard: Kubernetes has become the de facto standard for container orchestration, offering a vast ecosystem and community support.
    - Scalability and Resilience: Its inherent design principles of declarative configuration, self-healing, and automatic scaling are crucial for building resilient and scalable microservices and integration systems.
    - Automated Operations: Automates deployment, scaling, load balancing, and rollout/rollback of applications, significantly reducing operational overhead.
    - Cloud Agnosticism: While often used with cloud providers (GKE, EKS, AKS), Kubernetes can also be deployed on-premises or in hybrid cloud environments, providing flexibility in infrastructure choices.

---

9. Deployment & Scalability Considerations

Initial Phase:

- Deployment Strategy: Deploy the core services (API Gateway, Frontend, Mahak Sync Worker, PostgreSQL, RabbitMQ) as Docker containers.
- Environment: Utilize Docker Compose for local development and a single-server deployment (e.g., a powerful VM or bare-metal server) for initial production or staging environments. This simplifies initial setup and management.
- Database: PostgreSQL can be run as a Docker container or as a managed service. For the initial phase, running it within Docker on the same server or a dedicated small VM is feasible.
- Message Queue: RabbitMQ can also be deployed as a Docker container.
- Object Storage: MinIO can be deployed as a Docker container.

Scaling Phase:

- Orchestration: Migrate the Docker containers to a Kubernetes cluster. This is the primary step for enabling scalability and high availability.
    - Managed Kubernetes Services: Utilize cloud provider offerings like Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service

(EKS), or Azure Kubernetes Service (AKS) for a managed control plane and easier infrastructure management.
    - On-Premise/Edge Kubernetes: For on-premise deployments or environments with specific network constraints, lightweight Kubernetes distributions like K3s or Rancher can be considered.
- Scalability:
    - Horizontal Pod Autoscaling (HPA): Configure HPA for stateless services like the API Gateway and Mahak Sync Worker to automatically scale the number of pods based on CPU or memory utilization.
    - StatefulSets for Databases: Use Kubernetes StatefulSets for PostgreSQL to manage stateful applications, ensuring stable network identities and persistent storage. Leverage PostgreSQL clustering or replication solutions (e.g., Patroni) for high availability.
    - RabbitMQ Clustering: Deploy RabbitMQ in a clustered configuration across multiple nodes for increased throughput, fault tolerance, and availability.
    - Message Queue Consumers: Scale the number of consumers consuming from RabbitMQ queues to match the processing load. Kubernetes can manage this scaling.
- High Availability:
    - Multiple Replicas: Run multiple replicas of critical services (API Gateway, Sync Worker) to ensure that if one instance fails, others can take over.
    - Pod Anti-Affinity: Configure Pod anti-affinity rules to ensure that replicas of the same service are scheduled on different worker nodes, preventing a single node failure from taking down the entire service.
    - Persistent Storage: Ensure robust persistent storage solutions are used for PostgreSQL, backing them up regularly.
    - Load Balancing: Kubernetes services and ingress controllers automatically handle load balancing across healthy pods.
- Disaster Recovery: Implement backup and restore strategies for PostgreSQL and potentially message queue data. Consider multi-region deployments for critical services if high availability needs extend to data center failures.
- Monitoring and Alerting: Integrate comprehensive monitoring (Prometheus, Grafana) and alerting systems to track resource utilization, application

performance, error rates, and queue depths, allowing for proactive issue resolution and capacity planning.

End of Document