Defense Paper – Resilient Microservice Architecture for B2B Payment Processing

Table of contents:

1. Introduction

In today's enterprise landscape, large organizations increasingly rely on distributed, event-driven architectures to enable scalability, resilience, and maintainability. Traditional monolithic systems, while simpler to deploy, often fail to meet modern business demands for flexibility, fault isolation, and continuous delivery. This project explores the design of a **resilient cloud-native microservice architecture** tailored for a **B2B payment processing workflow** that integrates with partner SAP systems.

The system is designed as if it were to operate in a real-world production environment, though it remains a conceptual prototype for this internship. Its main objective is to ensure **reliability, data consistency, and operational transparency** across asynchronous and potentially unreliable system interactions — especially between external B2B partners and SAP back-end systems.

The proposed solution builds upon the principles of **loose coupling**, **event-driven communication**, and **resilience by design**, leveraging managed services conceptually aligned with AWS infrastructure components (e.g., API Gateway, Lambda, SQS, RDS, CloudWatch, EventBridge).
The project follows the key objectives of the internship:

1. Design a realistic, production-ready architecture that can withstand system failures.

2. Implement reliability mechanisms such as retries, outbox patterns, circuit breakers, and dead-letter queues.

3. Provide observability through structured logging, metrics, and tracing.

4. Define runbooks and failure recovery procedures for critical scenarios such as SAP downtime.

2. Problem Framing

The primary challenge addressed by this project is the **coordination between external B2B partners and internal SAP systems**, where **partner requests must be processed reliably even if downstream systems (SAP) are unavailable**.

In the actual enterprise environment, the integration between partners and SAP is fragile due to several constraints:

- The company acts as a middleware between multiple partners and their respective SAP instances.

- Partners often maintain their own SAP configurations, which are not directly accessible to our systems.

- SAP systems may become overloaded or unreachable, causing transaction failures or inconsistencies.

- Incoming payment requests must never be lost or processed twice, even under partial failures.

These challenges necessitate an architecture that can **defer work, handle transient errors gracefully, and maintain data integrity** across multiple asynchronous components.

From a reliability engineering perspective, the system must fulfill the following guarantees:

- **Exactly-once semantic processing**: the same payment should never reach SAP twice.

- **Durable persistence**: once accepted, data must not be lost.

- **Graceful degradation**: if SAP is down, the system continues to accept and queue requests.

- **Observability and traceability**: operators must be able to track every transaction and its status.

---

3. Architecture Defense

3.1. Overview and Design Principles

The architecture is based on a **microservice ecosystem** that separates concerns by domain. Each service is responsible for a single task, following the **Single Responsibility Principle (SRP)**. Services communicate **asynchronously** via message queues (SQS) and publish domain events (e.g., PaymentCreated, PaymentSettled, PaymentFailed) that other components can consume without direct dependencies.

The core services include:

- **Gateway** – Receives external API calls, authenticates partners and stores initial requests .
- **Request Dispatcher** - Acts as a state machine, it redirects the incoming requests to the corresponding services with the only exception for GET-Requests. In such a case it returns the status of a payment from the Payments Core Database.
- **Event Dispatcher** – Periodically polls the outbox table and publishes events to corresponding queues. It is actively triggered by the Event Dispatcher SQS only when the threshold of validated payment requests is reached in order to avoid busy polling and reduce cost.
- **Validator**- It processes incoming payment POST-Requests, enforces idempotency and publishes messages to the Event Dispatcher SQS.
- **Worker / SAP Adapter** – Processes validated requests, transforms data, and communicates with SAP.
- **Webhook Service** – Sends event updates (success/failure) back to partners via their configured webhooks.
- **Logger** – An AWS Lambda that acts as a transfer service for all logs to the CloudWatch and other Log- / Analytical platforms.
- **Health Checker** - An AWS Lambda that periodically checks SAP's status and adjusts the corresponding metrics on the CloudWatch platform.
- **Circuit Controller** - An AWS Lambda that implements a circuit breaker pattern depending on the corresponding CloudWatch metrics when SAP is under heavy load or unreachable.
- **Partner Config Service** – A relational data store (RDS) that holds partner-specific metadata (credentials, webhook URLs, mapping rules).
- **Payments Core** – A relational data store (RDS) that holds payments and outbox events for reliable message delivery.
- **Payment POST Request SQS** - An AWS SQS that holds incoming payment post requests of the partners that are yet to be validated by the SAP service layer. The Request Dispatcher publishes incoming requests to this queue and the queue itself invokes the Validator Lambda.
- **Worker SQS** - An AWS SQS that holds already validated payment requests that are yet to be processed by the SAP service layer. The Event dispatcher

Lambda publishes messages to this queue and the queue itself invokes the Worker service.
- **Webhook SQS** - An AWS SQS that holds partner notification messages that are yet to be delivered to the partner webhook endpoints. The Event Dispatcher service publishes messages to the queue and the queue itself invokes the Webhook Service.
- **Event Dispatcher SQS** - An AWS SQS that acts as a mechanism to actively poll the Payments Core RDS outbox tables for payment requests and notification messages so that a busy polling is avoided. The Worker Service publishes messages to this queue and after a certain threshold is reached the queue invokes the Event Dispatcher service.
- **Logger SQS** - An AWS SQS that holds all Log Data until it is processed by the Logger Service. Every other service of the system writes log messages to the queue and the Logger Service processes these messages. The queue ensures that no log messages are lost on the way to the Logs- / Analytics platforms.

All data flows are **event-driven**, ensuring that no service directly depends on another. This decoupling improves scalability and fault tolerance. If any downstream service (e.g., SAP) becomes unavailable, upstream services remain functional and continue queuing events for later processing.

3.2. Reliability Mechanisms

To achieve resilience, the architecture implements several well-known **resilience patterns**:

- **Outbox Pattern**: Guarantees reliable event delivery by storing domain events in the same transactional boundary as the payment data.

- **Retry and Exponential Backoff**: Worker Lambdas retry transient SAP calls while avoiding overload through exponential backoff.

- **Dead Letter Queues (DLQ)**: Messages that repeatedly fail are automatically routed to DLQs for later analysis or replay.

- **Circuit Breaker**: CloudWatch monitors SAP response metrics. If failure thresholds are exceeded, the circuit transitions to an “Open” state, pausing SAP-bound requests and preventing cascading failures. Once SAP stabilizes, the system reopens gradually through a “Half-Open” phase.

- **Idempotency and Unique Constraints**: Prevent duplicate payments and ensure deterministic behavior on retried operations.

These mechanisms collectively ensure that **no data is lost, duplicated, or indefinitely blocked**, even when subsystems fail.

3.3. Observability and Failure Handling

Observability is implemented through three core pillars:

1. **Metrics** – Latency, queue depth, retry count, and DLQ size are continuously monitored via CloudWatch or Prometheus-style dashboards.

2. **Logs with Correlation IDs** – Every request carries a correlation ID propagated across services for end-to-end traceability.

3. **Distributed Tracing** – Enables visibility into multi-service workflows (e.g., PaymentCreated → Worker → SAP → Webhook).

For **failure handling**, the system defines structured **runbooks**, particularly for:

- **SAP Down**: Circuit breaker opens; requests are queued; operators are notified; SAP health checks trigger recovery.

- **DLQ Replay**: Failed messages are re-queued automatically or manually once the root cause is resolved.
  These runbooks bridge the gap between automation and human oversight, ensuring quick recovery during incidents.

3.4. Justification of Architectural Choices

The selected architecture strikes a balance between **resilience, simplicity, and scalability**:

- **Event-driven design** ensures loose coupling and fault isolation.

- **Serverless compute (Lambda)** allows horizontal scalability without infrastructure management.

- **Queue-based buffering** shields upstream systems from overload and enables backpressure control.
- **Cloud-native services** (SQS, RDS, EventBridge, CloudWatch) simplify reliability implementation by leveraging managed capabilities.
- **Partner Config separation** enforces data ownership boundaries, aligning with best practices in B2B integrations.

This architecture, while designed for educational purposes, closely mirrors production-grade systems used by large enterprises handling mission-critical integrations.

---

4. System Flaws and Architecture Issues

4.1 Excessive Service Granularity

A notable drawback of the current architecture draft is its **over-fragmentation into too many small services**. The system comprises eight distinct microservices, five SQS queues, and two RDS databases, yet collectively serves only a single functional purpose — payment processing.

While this approach adheres strictly to the principles of microservice design and separation of concerns, it introduces unnecessary complexity at the operational and architectural level. Maintaining numerous small services increases the cognitive load on the architecture owner and complicates system evolution.

If additional business processes such as **order management** or **payment cancellation** were to be introduced, the architecture would require the creation of multiple new microservices, each with its own infrastructure components (e.g., dedicated queues, database schemas, and monitoring configurations). This would lead to **resource proliferation** and an exponential increase in maintenance effort.

Although the existing logging and correlation mechanisms ensure traceability across services, the overall manageability of the system remains challenging. A more pragmatic approach would be to **aggregate functionally related services** into consolidated components, even if this implies a slight deviation from the **Single Responsibility Principle (SRP)**.
In this context, such controlled violations of SRP are justifiable, as they would simplify the overall architecture, reduce operational overhead, and improve the clarity of ownership within the system.

5. Conclusion

The proposed microservice architecture demonstrates how **resilient design patterns and event-driven principles** can significantly improve reliability in distributed payment processing.
By separating services by responsibility, introducing asynchronous communication, and embedding resilience mechanisms at multiple layers, the system ensures operational continuity even under partial failures such as SAP downtime.

Furthermore, the incorporation of observability and automated runbooks transforms reliability from a reactive to a proactive discipline.
While this architecture is not deployed in production, it provides a **scalable, fault-tolerant blueprint** that can be directly applied to real-world enterprise environments.