**Check-in Service System Design Document**

This document outlines the design and implementation of a robust Check-in Service for factory worker attendance tracking. The system ensures reliable recording of clock-in/clock-out events while gracefully handling unreliable downstream systems through an event-driven architecture with comprehensive failure recovery mechanisms.

**1. The Challenge**: We need to handle two very different types of work:

1. Fast & Critical: Card readers need immediate response - employees can't wait
2. Slow & Unreliable: Legacy systems and emails can take seconds or fail entirely

**Our Solution**: Separate these concerns using an event-driven approach.

Card Reader → REST API (Fast) → Database → Kafka → Consumers (Slow) → External Systems

**2. How We Meet Each Requirement**

**2.1 Single REST Endpoint**

**Why one endpoint?** Card readers are simple devices - they just need to send "employee tapped card".

**How it works:**

1. Employee taps card → POST /api/attendance {employeeId: "123", eventType: "CHECK\_IN/CHECK\_OUT"}
2. System checks: Are they checked in? If not, create new shift. If yes, check them out.
3. Immediate response: "Success" or "Error" within milliseconds

**2.2** **Separating User Actions from Third-Party Calls**

**The Problem:** If we called legacy systems directly:

1. Card reader waits 30 seconds for slow legacy API → employee confused, queue forms
2. Email fails → card reader shows error even though check-out was recorded

**Our Solution:** "Fire and forget"

1. Sync: Record check-out in database → immediate success to card reader
2. Async: Publish "check-out completed" event to Kafka
3. Background: Separate services handle legacy system & email independently

**2.3** **Handling Legacy System Issues - Expecting Failure**

**Our Multi-Layer Approach:**

1. Retry with backoff (wait 3s, then 6s, then 12s...)
2. Dead Letter Queue for manual fixing
3. Nightly batch process as final backup (batch processing)

**Dealing with Rate-Limits and Legacy system outages**

**Rate Limits**

1. Consumer throttling: Limit concurrent requests per consumer
2. Exponential backoff: Increase delays between retries (3s → 6s → 12s)
3. Circuit breaker: Temporarily stop calling overloaded API

**Outages**

1. Kafka as buffer: Messages accumulate in topic during downtime
2. Health checks: Monitor external system status before sending
3. Batch fallback: Nightly job catches all unreported shifts

**2.4 No Lost Events - Multiple Safety Nets**

**Why we need this**: Factory payroll depends on accurate hours. Missing one check-out could mean missing 8 hours of pay.

**Our Safety Nets:**

1. **Database First**: Event saved before anything else
2. **Kafka Persistence**: Messages survive system restarts
3. **Batch Verification**: Nightly job finds any missing reports
4. **Manual Recovery**: Operations team can fix anything that slips through

**Why Batch Processing?**

**What it does**: Nightly job queries the database for any shifts that weren't successfully reported to the legacy system or where emails failed. So even if real-time processing fails completely for a day, the batch job at 10:30 PM will find and report all missing hours and no worker loses pay.

**4. Error Handling**

Real-World Failure Scenarios

|  |  |  |
| --- | --- | --- |
| **Scenario** | **What Happens** | **User Impact** |
| Slow Legacy API | Retry with increasing delays | None - employee already got success |
| Legacy API down | Messages queue up, batch handles later | None for hours, ops team alerted |
| Email service down | Retries, then DLQ | Employee gets daily summary instead |
| Database down | Card reader shows error | Employee tries again later |

**5. Monitoring & Tracing**

|  |  |  |
| --- | --- | --- |
| **Metric** | **Why It Matters** | **Good Value** |
| API response time | Card reader responsiveness | < 200ms |
| Kafka consumer lag | Are we falling behind? | < 100 messages |
| Legacy API success rate | Is external system healthy? | > 95% |
| DLQ size | Need human intervention? | 0 |

1. Distributed tracing with correlation IDs tracks each message from

card reader → database → Kafka → consumers → external systems.

1. Metrics (API latency, consumer lag, DLQ size) and alerts notify of issues
2. Structured logs with trace context provide full audit trail.

**6. Message Format Evolution**

1. Avro Schema Registry manages versioned schemas.
2. New fields are added as optional with defaults, ensuring backward compatibility.
3. Old consumers ignore new fields, new consumers handle both formats.
4. Zero-downtime deployments allow gradual rollout without message loss.