# Designing a Real-Time Buffer Monitoring System for Large-Scale Warehouse Operations

## A Case Study in Serverless Architecture, Decision Automation, and Infrastructure as Code

**Ignacio Balasch Sola**
ignacio@balasch.es

September 2025 — February 2026

*This report describes architectural patterns and engineering methodology.*
*Proprietary implementation details have been abstracted for confidentiality.*

**Abstract**

Over a six-month engagement at a major e-commerce company, I designed and deployed a **serverless, event-driven surveillance system** that monitors operational buffer health across a **large network of robotized warehouses** in real time. The system replaced manual monitoring performed by hundreds of operations specialists—who collectively spent a significant fraction of their working time on repetitive dashboard checks—with an automated pipeline that delivers actionable alerts within **∼1 minute** (vs. 15–30 minutes manually), achieving an **80% reduction in manual monitoring effort**.

The architecture leverages cloud-native services (Lambda, Step Functions, EventBridge, S3, Redshift) with complete **infrastructure as code**, enabling reproducible deployments across isolated development and production environments in under 10 minutes. The system is **horizontally scalable**—onboarding a new warehouse requires a single database statement and no code changes.

This report presents the engineering methodology, architectural decisions, and lessons learned—focusing on transferable patterns rather than proprietary implementation details. Although I contributed to other projects during my time at the company, this system represents the core of my work and the largest individual deliverable.

# 1 Introduction

## 1.1 The Domain

In large-scale e-commerce warehousing, packages move through sequential processing stages—picking, sorting, and packing. Between each stage, items accumulate in **physical buffers** (conveyors, lanes, drop-zones) that decouple upstream and downstream processes operating at different speeds. When buffers approach capacity (*overflow*), congestion cascades through the processing chain; when they deplete (*starvation*), downstream workers and equipment sit idle. Both conditions threaten customer delivery promises.

## 1.2 The Problem: Manual Monitoring at Scale

Across a European network of robotized warehouses, **hundreds of operations specialists** spend a **significant fraction of their working time** on buffer monitoring—integrating data from multiple disconnected systems, mentally aggregating signals, and repeating this analysis every 15–30 minutes throughout their shifts. The cognitive burden is substantial: each specialist must navigate multiple dashboards, perform manual calculations, and remember shift schedules and warehouse-specific patterns. This doesn't scale—adding warehouses requires proportionally more specialist time, and the 15–30 minute cycle means problems are often detected too late for effective intervention.

## 1.3 The Mandate

I was tasked with automating this workflow. The core requirements: alert within 5 minutes of a problem (vs. 15–30 min manual), achieve >90% detection accuracy, scale horizontally across the warehouse network, deliver specific quantified recommendations (not just alerts), and maintain full decision transparency for operations team trust.

## 1.4 My Starting Point

I began as a BI intern with strong SQL and Python skills but **zero experience** with cloud infrastructure, serverless architecture, or production systems. Every concept in the tech stack—Lambda, Step Functions, EventBridge, VPC networking, IAM, CDK—was learned during the project. This shaped many architectural decisions: I was building a production system while simultaneously learning the paradigms it required.

# 2 System Architecture

## 2.1 Pipeline Overview

The system consists of six sequential stages orchestrated by a state machine:

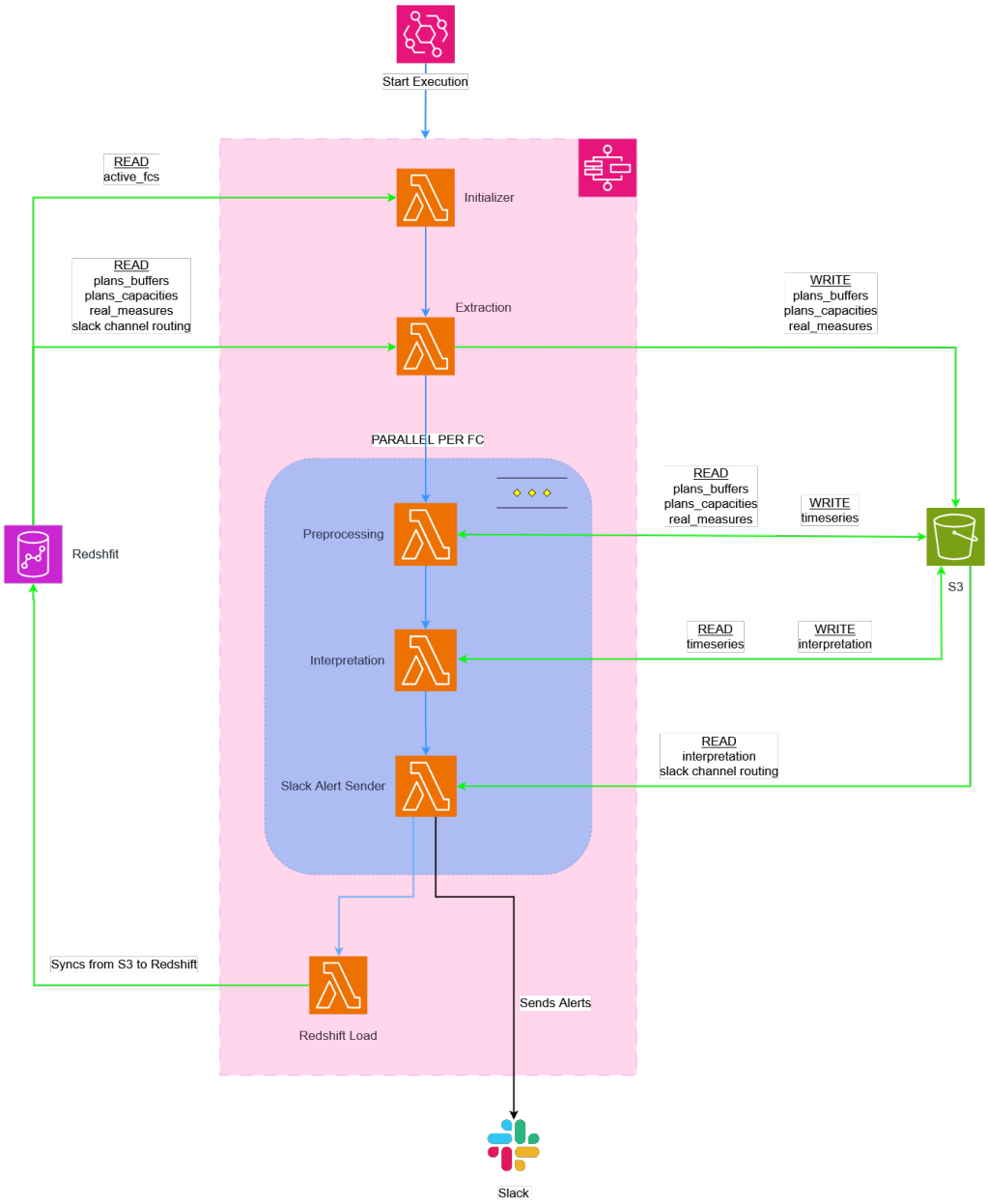


Figure 1: Pipeline execution flow — global steps run once, per-warehouse steps execute in parallel

The pipeline follows a global-then-parallel pattern. **Step 0** generates a deterministic snapshot timestamp and loads the active warehouse list. **Step 1** extracts data from all warehouses in a single batched query per source (with four queries running in parallel threads), then partitions results in-memory—dramatically reducing database load compared to per-warehouse querying. **Steps 2–4** execute in parallel across warehouses: preprocessing raw data into unified timeseries, interpreting anomalies into recommendations, and delivering formatted alerts. **Step 5** runs after alerts are delivered and syncs all artifacts back to the data warehouse for dashboards and historical analysis.

# 3 Data Pipeline: From Raw Signals to Unified Timeseries

## 3.1 The Integration Challenge

The system consumes data from three upstream systems, each owned by different teams. These systems use different data models, refresh at different rates, and provide overlapping but not identical coverage. The preprocessing pipeline's job is to unify them into a single, consistent timeseries.

## 3.2 The Operational Unit Abstraction

Perhaps the most important conceptual contribution of this project was defining **Operational Units**—the atomic level at which the system evaluates buffer states and recommends actions.

**The problem without this abstraction**: Warehouses have dozens of individual process paths (different conveyor routes, sorting lanes, packing stations). Without clear aggregation rules, the system would produce fragmented, potentially conflicting recommendations for related paths that share the same control mechanisms.

**The solution**: Each operational unit represents a distinct operational concern with clear scope, defined actions, and independent evaluation.

**Key design decision**: Some process paths contribute to multiple operational units. For example, an induct path contributes both to the aggregate total picking buffer *and* to its own induct-specific unit. This intentional duplication ensures that aggregate staffing decisions and induct-specific throttle decisions can both be evaluated correctly, with conflict resolution handled in a later substep.

**Standardized identifiers**: Every operational unit has a deterministic, human-readable ID following the pattern `{Warehouse}_{Workpool}_{PathOrTotal}`. These IDs serve as primary keys across all pipeline stages, enabling consistent tracking from preprocessing through interpretation to delivery.

## 3.3 Building a Fast Iteration Loop

Getting the correct data was one of the biggest challenges of the project. I couldn't speak directly to upstream table providers, and documentation was sparse or nonexistent. The only viable validation method was to extract data, process it, and show the results to a domain specialist quickly—before the data became stale and unverifiable.

This forced me to learn AWS Lambda deployment and dashboard development out of necessity. Eventually, I built a pipeline that would extract data, centralize it in intermediate storage with in-between visualizations, and let specialists validate the methodology through live dashboards. Once that feedback loop was in place, iteration speed increased dramatically. **Building the infrastructure to iterate fast was as important as the system itself.**

## 3.4 Normalization Can Wait for the MVP

I approached the MVP with a strong desire to follow best practices, including full data normalization. Large businesses suffer from messy data warehouses, and I wanted to avoid contributing to that problem. But I learned that normalization adds friction during the exploratory phase—when schemas are changing daily and the goal is validation, not durability. The better approach is to develop with a simpler schema, validate the solution, and then normalize once the data model stabilizes.

# 4 Interpretation Algorithm

The interpretation algorithm processes recommendations through three sequential layers, each adding progressively more context. This decomposition keeps each layer simple, independently testable, and separately debuggable—every substep writes its output to S3, so you can inspect the recommendation at any stage.

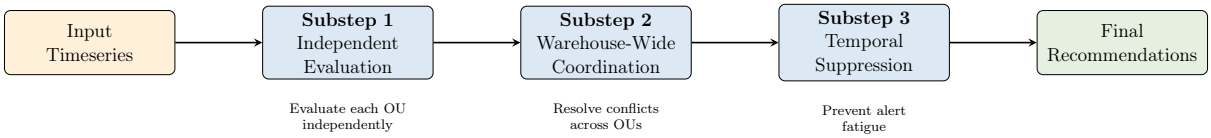


Figure 2: Three-substep interpretation: progressive context enrichment

In **Substep 1**, each operational unit is evaluated independently against its own timeseries. The algorithm routes each unit to one of three decision branches based on type, applying hardcoded threshold checks and trend analysis. The logic is deliberately simple and fully inline—every decision can be traced by reading a single file from top to bottom.

**Substep 2** resolves conflicts across operational units within the same warehouse. Some recommendations are mutually exclusive—for example, a broad throttle affecting all picking operations subsumes a narrower induct-specific throttle. The substep detects these conflicts and suppresses the narrower action, preserving the original recommendation in the explanation field for auditability.

**Substep 3** prevents alert fatigue by applying temporal suppression. Without it, the system would re-alert every 5 minutes for ongoing problems. The substep loads 60 minutes of historical recommendations from S3 and applies action-specific suppression windows—shorter for staffing adjustments, longer for capacity rebalancing, and none at all for emergencies. Each recommendation is labeled as `NEW`, `ONGOING`, `CLEARED`, or `SUPPRESSED`.

## 4.1 Finding the Right Level of Abstraction

The interpretation algorithm went through three major iterations before I found the right level of abstraction. First, I designed a system to identify hardcoded circumstances stored in configuration tables, then map from circumstances to actions—the goal was to collect structured data that could later be used to apply machine learning and fine-tune the algorithm automatically. Then, I tried enriching the timeseries with computed signals and windowed aggregations, feeding them into an abstract framework that would determine actions based on configurable combinations of signals and windows.

Neither approach was *wrong*—both worked, and the former structure is likely to be built in the next iteration. But I ended up pivoting to something far simpler: a hardcoded, inline decision tree that I could present to stakeholders and explain exactly how every decision was made. **In hindsight, the MVP should have been scoped to this simple version from the start.** The more sophisticated architectures were valid future iterations, not MVP requirements—and conflating the two cost time that could have been spent shipping and learning from real feedback sooner.

# 5 Infrastructure as Code

The entire system—Lambda functions, state machine, scheduling rules, VPC networking, IAM policies, storage lifecycle rules—is defined in code using AWS CDK (Cloud Development Kit), developed in collaboration with a senior infrastructure engineer. This means the complete system can be rebuilt from version control in roughly 10 minutes, any engineer on the team can deploy without specialized knowledge, and every infrastructure change is tracked in Git history alongside the application code.

The same codebase deploys to both development and production environments. A single environment variable, injected at deployment time, controls storage paths, database schemas, alert routing, and message verbosity—eliminating the "works in dev, breaks in prod" failure mode entirely.

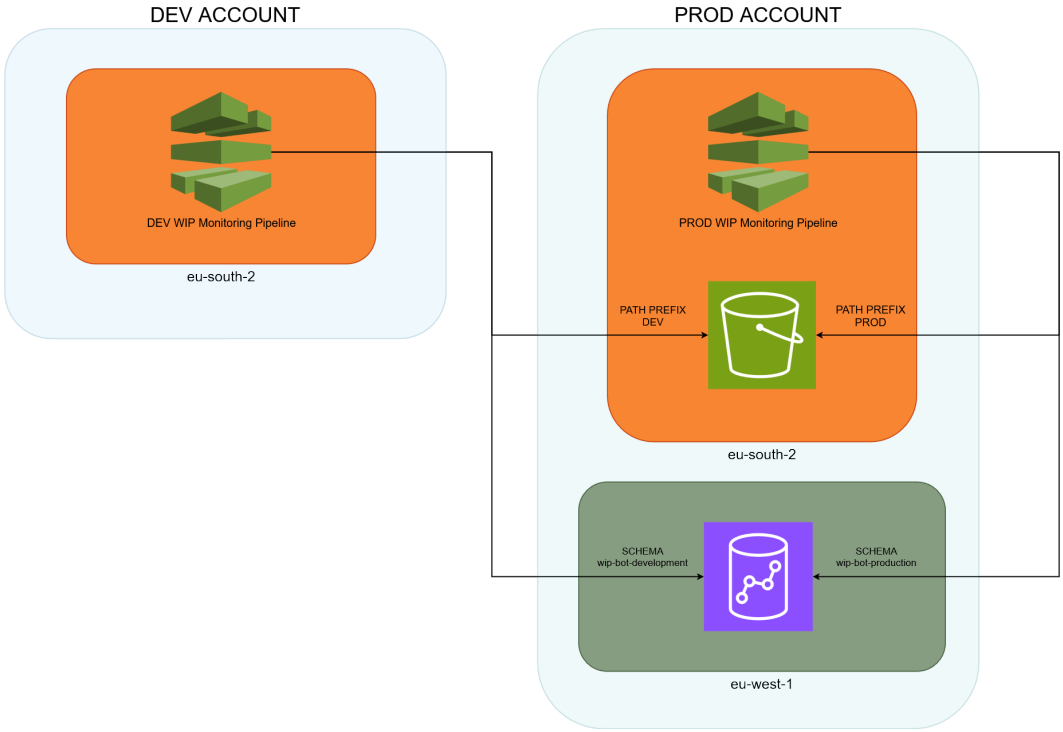


Figure 3: Multi-account architecture with environment isolation — shared storage (path-based separation) and shared data warehouse (schema-based separation)

Both environments share a single object storage bucket and a single data warehouse cluster, with path-based and schema-based separation respectively. This provides complete logical isolation without duplicating infrastructure. A practical challenge was that the data warehouse resided in a different AWS region than the pipeline infrastructure (for regulatory compliance), requiring explicit cross-region configuration—a non-obvious requirement that initially manifested as silent failures. Similarly, the data warehouse sits inside a private network (VPC), so Lambda functions must use private IP addresses rather than public DNS—navigating security groups, subnets, and private endpoints was a significant learning curve.

One area I wish I had invested more effort in was local testing. Every code change required a commit-and-push cycle to test in the development environment, creating friction in the development loop. A local emulation layer would meaningfully accelerate future development velocity.

# 6 Alert Delivery

The delivery layer is a pure presentation component with no interpretation logic. It reads final recommendations from S3, formats them as rich messages, and posts them to the appropriate team channels. In development mode, all operational units are shown (for algorithm validation); in production mode, only action-required units appear—if everything is healthy, no message is sent at all. Each alert includes a pre-filled feedback link so operations teams can report whether the recommendation was correct, creating a continuous improvement loop. Alert routing is self-service: operations teams manage it through simple database statements, with changes taking effect on the next 5-minute cycle—no code changes or deployments required.

## 6.1 The Missing Piece: Closing the Feedback Loop

One of the most impactful improvements I can envision for this system is closing the feedback loop—routing structured operator responses from the messaging platform back into the data warehouse. Today, feedback links exist in every alert, but the responses live in an external form tool, disconnected from the pipeline's data.

If feedback flowed back into the same warehouse where recommendations are stored, entirely new capabilities would open up. The system could automatically cross-reference operator judgments with its own outputs to measure correctness per warehouse, per operational unit type, and per algorithm version. New interpretation algorithms could be tested in the development environment and automatically promoted to production once feedback data confirms they outperform the current version. Feedback patterns could reveal systematic blind spots—categories of problems the algorithm consistently misidentifies—enabling targeted improvements rather than global threshold tuning. Over time, this creates a virtuous cycle: better algorithms produce better recommendations, which earn more operator trust, which generates higher-quality feedback, which enables even better algorithms.

This is, in my view, the single highest-leverage improvement remaining in the system's roadmap.

# 7 Conclusion

This project was a privilege. Over six months, I went from a business intelligence intern with no cloud experience to someone who designed, built, and deployed a production-grade serverless system monitoring a large warehouse network in real time.

The system delivers measurable impact: an 80% reduction in manual monitoring effort, detection times compressed from 15–30 minutes to roughly one minute, and a horizontally scalable architecture that grows with the network without proportional cost.

But the most valuable outcomes weren't technical. They were the lessons about building systems in the real world—where data is messy, requirements evolve, stakeholders have limited bandwidth, and the best architecture is the one that ships and earns trust. I learned that pragmatic simplicity beats theoretical sophistication, that fast iteration loops matter more than elegant designs, and that constraints aren't obstacles—they're design inputs that make the final system better.

I'm proud of what I built, and grateful for the opportunity to build it.