Allianz Project Log

Kafka Connect Operations -
Customer-Specific Requirements

Project: Allianz EBS @ Confluent Cloud

Focus: Establish Self-Service Operations for Kafka Connect Connectors in the context of access restrictions.

Author: Dr. Mirko Kämpf

Date: 2025-12-16

Version: 1.0

Status: Draft

Executive Summary

This document captures the operational model for Kafka Connect in a **three-party shared infrastructure** where:

- **EBS Infrastructure Team** manages the Confluent Cloud account and network connectivity
- **Alliance Department Teams** (customers) need to deploy and manage connectors
- **Confluent Cloud** provides the managed platform

The key challenge is enabling **self-service connector operations** while maintaining control, quotas, and security in a shared multi-tenant environment where end users **do not have Confluent Cloud UI access**.

---

1. Customer Context and Expectations

1.1 Background

The EBS team manages a shared Confluent Cloud environment for various internal clients ("Alliance entities"). These clients need Kafka Connect to integrate data sources (e.g., databases and object stores) with Kafka. The current operational model creates a bottleneck: end-users do not have direct access to the Confluent Cloud UI or permissions to deploy connectors themselves.

The current process requires opening a ticket with EBS, which is not scalable. EBS seeks a solution allowing clients to operate more autonomously while maintaining security and control over the shared infrastructure. They are familiar with Kafka Connect generally but lack specific knowledge of its operational aspects within Confluent Cloud, particularly around multi-tenancy, logging, monitoring, and resource management.

1.2 Critical Pain Points

Pain Point 1: Undefined and Unscalable Operational Model

Problem:

- End-users want to use fully managed connectors but **lack the necessary access and permissions** to the Confluent Cloud UI
- Current workaround: **open tickets with EBS team**, which manually handles deployment
- **Even 20 concurrent requests** would be problematic
- This central dependency creates a bottleneck and prevents user autonomy
- Roles and responsibilities for who operates Connect flows versus the underlying Connect cluster are unclear

Impact:

- Not scalable beyond a handful of users
- EBS team becomes a bottleneck
- Slows down all departments

Required Solution:

- Clear operational model with defined roles
- Self-service or semi-automated deployment process
- Scalable to 20+ concurrent departments

Pain Point 2: No Log or Metrics Visibility for End-Users

Problem:

- End-users **cannot access the Confluent Cloud UI**, so they are "driving blindly"
- Unable to see running status, failures, or root causes
- Troubleshooting becomes impossible for them and burdens the EBS team
- Logs are crucial for diagnosing configuration errors
- While EBS could retrieve log dumps, this is slow and inefficient
- Users need access to **metrics (CPU, memory) and logs (exceptions, behavior)** to operate effectively

Impact:

- Every connector issue requires EBS involvement
- Departments cannot self-troubleshoot
- Extended downtime when issues occur
- EBS team overwhelmed with support requests

Required Solution:

- Alternative mechanism to access logs (without UI access)
- Health check dashboard showing connector status
- Integration with Dynatrace for monitoring
- Logs exported from fully managed Kafka Connect

Pain Point 3: Uncertainty About Resource Management and Isolation

Problem:

- Shared, multi-tenant Confluent Cloud environment raises concerns
- Fear of "noisy neighbors"—e.g., one user deploying too many connectors and exhausting a shared VM's RAM
- Do not know what **resource management features, such as quotas** (e.g., connector limits per tenant), are available in Confluent Cloud
- Do not know how **workloads can be isolated** (e.g., dedicated vs. shared Connect clusters)
- Without clarity, one cannot design a robust, fair multi-tenant service

Impact:

- Risk of resource exhaustion
- One department could impact others
- Difficult to attribute costs
- Cannot guarantee SLAs per department

Required Solution:

- Understand Confluent Cloud quota capabilities
- Define isolation strategy (shared vs. dedicated)
- Implement quota enforcement mechanism
- Monitor resource usage per department

Pain Point 4: Unclear and Insecure Configuration Deployment Process

Problem:

- EBS team **cannot give customers direct API keys** to the Confluent account to run Terraform
- Shared clusters mean full API access would be too dangerous
- Need a secure process where customers submit connector configurations (e.g., via Git repo) and EBS runs the deployment
- This introduces extra steps and potential delays
- Full lifecycle—from configuration submission to deployment and health checks—must be defined securely and efficiently

Impact:

- Security risk if API keys are distributed
- Manual process if not automated
- Delays in deployment if EBS in the critical path

Required Solution:

- GitOps workflow for configuration submission
- Automated validation and deployment
- Restricted API keys with RBAC (if possible)
- Clear approval workflow for production

1.3 Customer Expectations

Expectation 1: Comprehensive Operations Handbook

What Customer Expects:

- Solution design or operations handbook outlining a clear, scalable process for internal clients to use Kafka Connect
- Detail **operational procedures for integrating departments**
- EBS is positioned as an infrastructure provider between Confluent Cloud and end-users

- Move from manual, ticket-based processes to an automated or self-service model
- Document describing **how customers can start and manage connectors**

Deliverable Required:

- Operational procedures document
- Step-by-step workflows with diagrams
- Role definitions and responsibilities
- Runbooks for common operations

Expectation 2: Independent Monitoring and Troubleshooting

What Customer Expects:

- Solution enabling clients to independently monitor and troubleshoot connectors
- Mechanisms to **access logs and view health status/metrics**
- Integration with existing monitoring tool, **Dynatrace**
- If direct UI access not possible, provide alternative methods:
    - Exporting logs from fully managed Kafka Connect
    - Building a health-check dashboard using the Connect API

Deliverable Required:

- Logging and monitoring strategy
- Health check dashboard
- Dynatrace integration plan
- Confluence engagement for the log export feature

Expectation 3: Clarity on Multi-Tenancy Capabilities

What Customer Expects:

- Clarity on Kafka Connect capabilities and limitations within Confluent Cloud for multi-tenancy
- Options for **resource isolation** (shared vs. dedicated clusters)
- Options for **resource management** (quotas on connectors or tasks)
- Guidance on preventing noisy neighbors
- Design a stable service with fair resource allocation

Deliverable Required:

- Multi-tenancy strategy document
- Quota enforcement mechanism
- Isolation decision matrix
- Monitoring for resource contention

Expectation 4: Leverage Native Confluent Tools

What Customer Expects:

- Use Confluent CLI where possible instead of building custom tools
- Architectural principle: "Leverage Confluent as much as possible."
- Open to giving users access to **Confluent CLI with restricted permissions**
- Balance user autonomy with EBS's control and security over central infrastructure

Deliverable Required:

- CLI-based workflow documentation
- RBAC configuration for restricted API keys
- Training materials for CLI usage
- Comparison of native vs. custom solutions

1.4 Additional Context

Existing Infrastructure and Practices:

- EBS team establishes network connectivity (Private Link) between end-customer data sources and Confluent Cloud cluster **on a case-by-case basis using terraform**
- Customer uses **GitOps for other configurations** (like topics) and is considering the same model for connector configurations
- Customer already forwards metrics from Confluent to **Dynatrace** via existing integration
- Similar approach could be built for logs (e.g., a Kubernetes pod reading from a Kafka topic and forwarding to Dynatrace)
- Terraform demo in progress to showcase end-to-end network connectivity steps (not completed yet, see inspection report)

Current Capabilities:

- Confluent Platform Metrics collection: ✅ Already integrated with Dynatrace
- GitOps workflows: ✅ Used for topic management
- CLI distribution: 🔶 Possible but not yet configured

Gaps:

- Network connectivity: 🔶 Process partially established (Private Link inbound, not yet outbound from Kafka Connect)
- Log visibility: ❌ Not available for end-users
- Quota enforcement: ❌ Mechanism not defined
- Connector deployment automation: ❌ Manual process currently

- Self-service access: ❌ No API keys for departments

1.5 Suggested Solutions

Our analysis from the workshops identified four potential solutions to address the pain points:

Solution 1: CLI-Based Self-Service with RBAC

- Provide end-users the Confluent CLI with restricted roles and service accounts
- Users manage only their own connectors (create, describe, delete) via CLI
- No UI or infrastructure access required
- Leverages native Confluent tooling
- Reduces custom development
- Gives users control and visibility (including log access via CLI)

Status: Documented in Solution Draft (SD) Section 3.1 (Option B) and Section 4.2

Solution 2: GitOps-Driven Automation

- Formalize a GitOps workflow
- Dedicated repo holds declarative connector configs (JSON/YAML)
- CI/CD pipeline triggers on commits
- Validates naming conventions, resource limits
- Uses service account to apply configs to Confluent Cloud via Terraform or CLI
- Yields an auditable, automated, controlled path

Status: Documented in SD Section 3.1 (Option A) and Section 4.1

Solution 3: Centralized Monitoring & Alerting Hub

- Solve visibility gap by sinking connector logs to dedicated Kafka topic
- Build lightweight service (Kubernetes pod) that consumes from this topic
- Forward logs to Dynatrace
- Create tenant-specific dashboards and alerts in Dynatrace
- Additionally, create internal dashboard using Kafka Connect REST API
- Display real-time connector status (`RUNNING`, `FAILED`)

Status: Documented in SD Section 5.3 (Three-Tier Logging Strategy)

Solution 4: Hybrid Model (CLI + GitOps)

- Use GitOps for initial connector provisioning (governance and proper setup)
- Allow users (with restricted service accounts) to use CLI for operational tasks:
    - Pause, resume, status, logs
- Balances centralized control with user agility
- Best of both worlds

Status: Documented in SD Section 3.2 (Hybrid Model - Recommended)