



SAP Incremental Ingestion Case Study

Table of Contents

Introduction - Current Challenges & Business Needs

Design Principles & Assumptions

Proposed Architecture

Implementation Approach

- ADF SAP CDC Connector Details
- Data flow: ADLS Bronze → Silver → Postgres DV2 (Hubs/Links/Sats) → BI Tableau
- How it stays reliable, secure, and low-cost
- SAP Connector Choices and tradeoffs

Data Quality & Governance

- Data Governance with Microsoft Purview

Risks & Mitigations

Summary & Next Steps

Appendix (Optional)

Introduction

Case Study Topic : *“SAP use case: loading huge data from SAP tables with pyRFC connection is less performant. pyRFC is deprecated and not supported from SAP anymore. What alternative solutions can you suggest to efficiently load incremental data from SAP?”*

Current Challenges

- **Scalability limits** → not scalable for TB scale data + 300 – 400 pipelines.
- **Performance bottlenecks** → pyRFC cannot handle large table volumes since it doesn't scale out.
- **Operational pain** → frequent failures, manual restarts, high maintenance.
- **Deprecation risk** → pyRFC no longer supported by SAP.

Business Needs and Desired State

- **Scalable, incremental, reliable, and future-proof ingestion**
- Must ensure: **trustworthiness, traceability, cost-efficiency**
- Critical for business → **timely BI insights & reporting**
- Designed to be **cloud-native & AI-ready**

Design Principles & Assumptions

Assumptions:

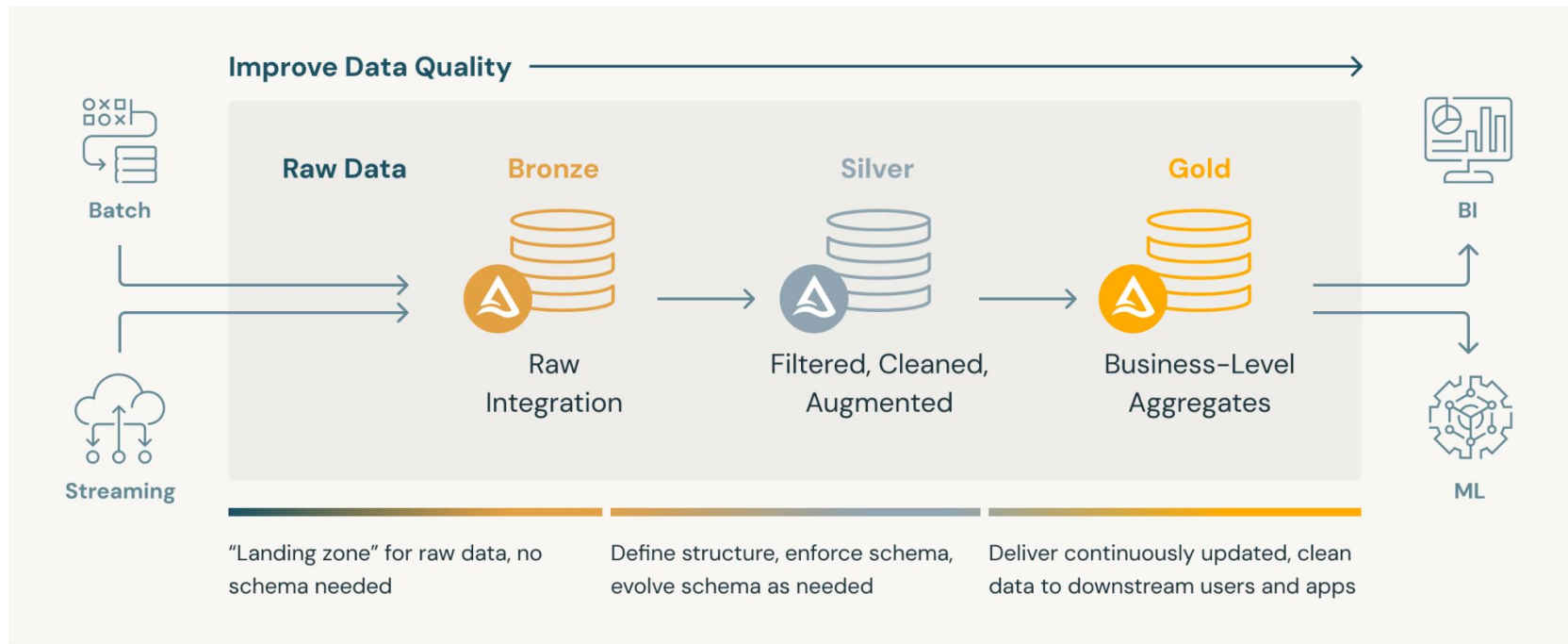
- SAP source = *S/4HANA or ECC*
- SAP ODP framework is possible or already enabled.
- Data scale = *10–20 TB variable* , 300–400 pipelines
- Target = *Postgres Data Vault 2.0 + Tableau BI (or any other BI Product)*
- Ingestion must be *incremental*, not full extracts
- It's an assumption based case study so there may be gaps if compared to current cosnova landscape.

Design Principles:

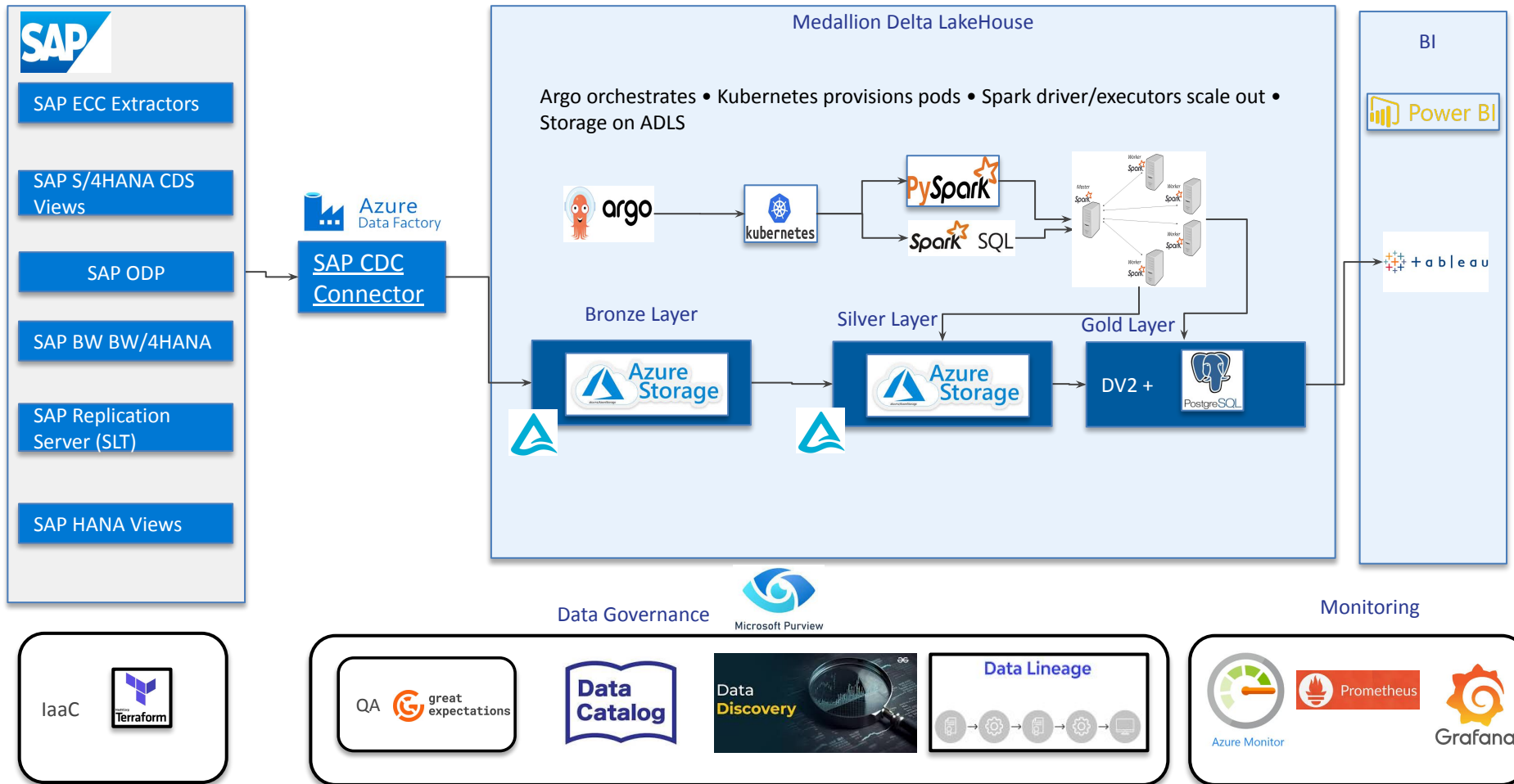
- **Scalable** → handle TB-scale data with growth in mind
- **Incremental-first** → CDC or delta-based loading
- **Cloud-native** → integrate with Azure ecosystem (ADLS, Purview etc.)
- **Reliable & Traceable** → monitoring, lineage, auditability
- **Cost-efficient** → avoid unnecessary full reloads
- **Future-proof** → AI-ready foundation for downstream use cases

Proposed Solution Architecture

Data Flow Designed with “Medallion Architecture”



Proposed Architecture - High Level - All Components



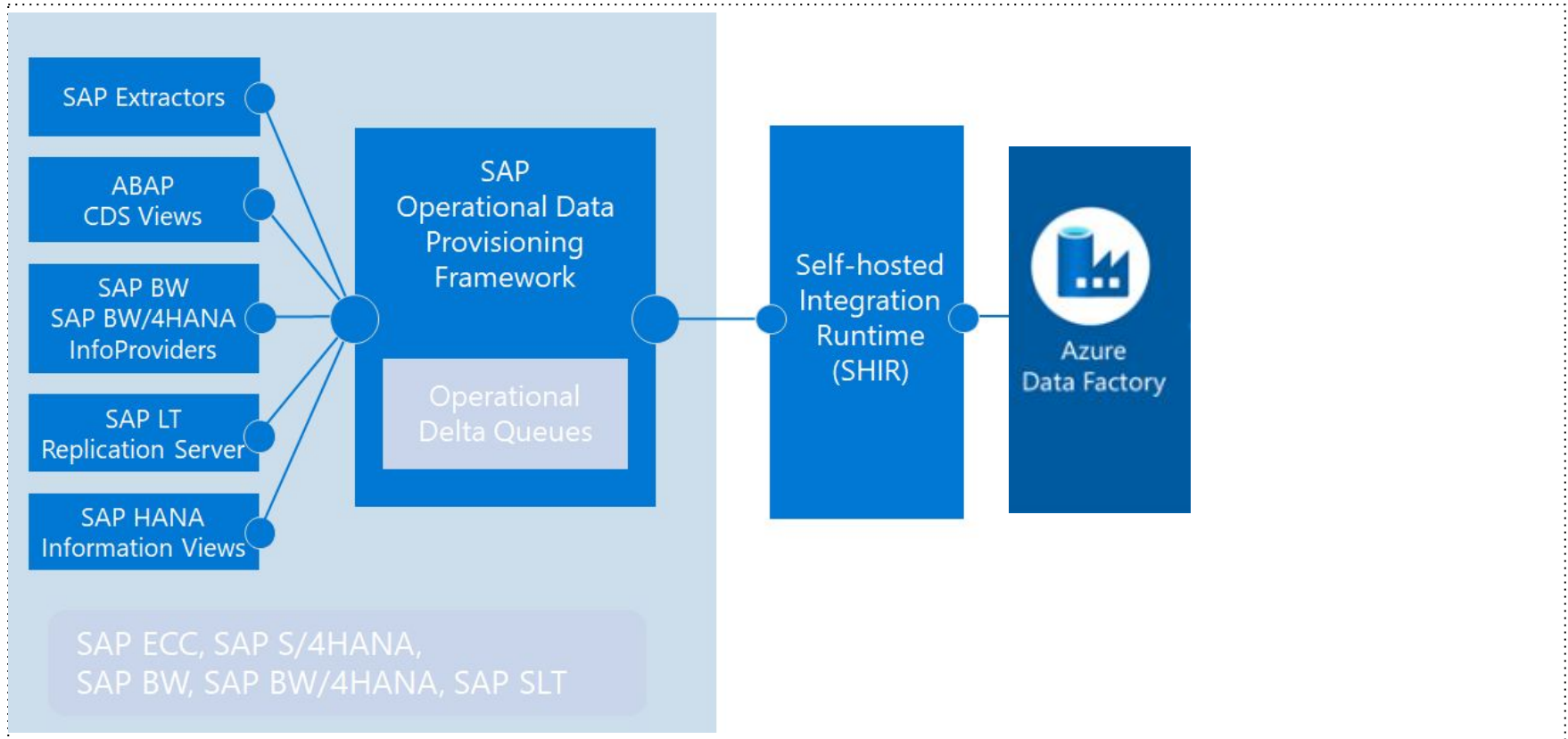
Why ADF SAP CDC Connector ?



Addresses Current Challenges & Meets Business Needs (Business come first , solution needs to tie back to business needs.)

- **Scalability** → Handles TB-scale data and 300–400+ pipelines via delta queues and parallelization.
- **Performance** → Near real-time ingestion without pyRFC bottlenecks.
- **Future-proof** → pyRFC deprecated → ODP-based connector is fully SAP-supported.
- **Operational efficiency** → Built-in checkpointing, idempotency, and automated retries reduce manual restarts.
- **Incremental & Reliable** → True CDC ensures only changes are moved, reducing cost & load.
- **Trust & Traceability** → Works seamlessly with Purview lineage + Great Expectations QA.
- **Timely BI** → Near real-time data flow into Synapse/Lakehouse → Power BI/ML-ready.
- **Cloud-native & AI-ready** → Deep Azure integration, scales elastically with SHIR + ADF.

ADF SAP CDC Connector Details



ADF SAP CDC Connector Details



ADF SAP CDC Connector – Key Points





- **Native SAP CDC** → Uses **ODP delta queues** for incremental extraction (low SAP impact).
- **Checkpointing & Idempotency** → Tracks **delta tokens** to resume safely, no duplicate loads.
- **Self-Hosted IR (SHIR)** → Required for SAP connectivity; can be **clustered for HA + scalability** (normally not needed as throttling happens at SAP Queue level to avoid SAP Performance issues.)
- **Flexible Modes** → Full, delta, or hybrid; support for **partitioned extraction** (e.g., fiscal year, company code).
- **CDC Metadata** → Operation type (I/U/D), sequence, timestamp → used for downstream merges.
- **Azure-Native** → Seamless with ADF pipelines, ADLS, Purview lineage, and Argo orchestration.



Risks & Mitigations

- **Single-node SHIR bottleneck** → Use **SHIR cluster** + partitioning.
- **SAP throughput limits** → Schedule extractions in **low-load windows**.
- **Schema drift** → Detected via **Great Expectations + Purview alerts**.

SAP Connector Choices and tradeoffs

Option	Pros	Cons	Fit for Use Case
ADF SAP CDC (ODP) 	Native CDC, Azure-integrated, minimal SAP load, retries , lineage , Scalability	Needs ODP setup in SAP , Needs SHIR Setup	✓ Primary choice
SAP BODS (Data Services) 	SAP-native ETL, mature	Extra license, on-prem infra, not cloud-native	✗ Heavy & costly
SLT (SAP Landscape Transformation Replication Server) 	True near-real-time, decoupled SLT → Event Hubs/Kafka → ADLS	Extra infra/ops, overkill if not sub-minute, Not suitable for our use case as requirement is incremental ingestion not real time.	⚠ Only for hot tables
Direct JDBC 	Simple, fallback	No CDC, high SAP load, fragile watermarking	✗ Last resort

Implementation: Data flow (Verbose, Pls be patient :)

- **SAP Change Capture (ODP)**
ODP delta queues (batch) expose table-level changes with sequence/order guarantees.
- **Ingestion via ADF SAP CDC Connector**
ADF pulls deltas incrementally, handles retries, and tracks progress (watermark/ODP token). Each record carries operation type (**_op: I/U/D**), source timestamp (**_cdc_ts**), and extraction batch (**_batch_id**).
- **Land to Bronze (ADLS, append-only)**
Write every CDC event **as-is** (no upserts). Include metadata columns: **_op, _cdc_ts, _batch_id, _source_system, _file_id**. Partition by business date (e.g., **ERDAT, GJAHR**) or ingestion date (**ingest_date**). Target 128–256 MB files. Bronze has all raw data. Format is Delta Table (ADC CDC can write delta). Subsequent Pyspark also writes delta parquet to realize scale and cost efficiency.
- **Orchestration & Compute (Argo → AKS → Spark)**
Argo submits a job; AKS spins up **driver + executor pods** (executors on spot pool). **Dynamic allocation + AQE** shrink/expand executors through the job.
- **Silver Quality Gate (Great Expectations)**
- **Great Expectations runs inside the PySpark job on AKS during the Bronze → Silver step**
Validate schema (no unexpected columns), required fields not null, type checks, and basic range checks. Failures go to **DLQ** (separate path + reason), successes proceed.

Implementation: Data flow

- **Silver Transform (idempotent merge)**
Read Bronze CDC; **dedupe** by primary key + **_cdc_ts** within a window; apply **business type casting & standardization**. Perform **idempotent MERGE** to build the latest state table; track **_valid_from** / **_valid_to** where needed.
- **History Handling (Slowly changing dimension)** At the **Silver** → **Gold step**, attributes that change over time (e.g., customer address, material description) are tracked using **Slowly Changing Dimension**. Instead of overwriting, we keep **historical versions** so reporting and audit can show how data looked at any point in time.
- **Referential Integrity & Enrichment**
 - Also in the **Silver** → **Gold step**, we enforce **referential checks** (e.g., every order links to a valid customer).
 - Gold tables are **enriched with small reference dimensions** (e.g., company codes, materials) so BI tools see business-friendly fields.
- **Promote to Gold (DV2 in Postgres)**
Load **Hubs (business keys)**, **Links (relationships)**, **Satellites (context/history)** with **hash keys (salted)**, **load_ts**, **record_source**. Build **Star Schema marts** (facts/dims) over DV2 for BI performance.

Implementation : Data flow

- **Consumption (Tableau / Power BI)**

BI refreshes incrementally from marts; aggregates and pre-computed KPIs reduce query cost. Sustainability/ESG metrics sourced from curated Gold.

- **Governance & Lineage (Purview)**

Assets auto-scanned; lineage stitched **SAP** → **ADLS Bronze** → **Silver** → **DV2/Stars** → **BI**. Azure AD policies and sensitivity labels enforce access.

- **Monitoring**

Azure Monitor/Log Analytics + **Prometheus/Grafana** track CDC lag, job duration, GE failure rate, executor utilization, spot evictions.

How it stays reliable, secure, and low-cost

- **Cost & Scale:** AKS **dynamic allocation** + **spot nodes**; scale-to-zero between runs; file compaction (128–256 MB),
- **Delta Table Shines** - format parquet - Strong SPARK compatibility makes it scalable and cost optimized - CODE OPTIMIZATION - PARTITION PRUNING / WRITE OPTIMIZE with Z-Ordering (multi-dimensional clustering)/ VACUUM / COALESCE . Real world example - 10TB table has avg select for ~5 seconds.
- **Resilience:** ODP **resume tokens**, idempotent **MERGE** in PySpark, backfill path for historical loads ADF CDC provides one time historical load and subsequent incremental load options.
- **Quality & Trust: Great Expectations** gates at Bronze/Silver; failures route to “Error Area” in silver zone and can be replayed.
- **Governance: Microsoft Purview** for catalog, search, and **end-to-end lineage** (SAP → ADLS → DV2 → BI) with Azure AD policies.
- **Monitoring: Azure Monitor + Log Analytics** and **Prometheus/Grafana** for CDC lag, job SLAs, and cluster health alerts.
- **Lifecycle rules in ADLS** - As per agreed days , Expire for Bronze and archive for silver and Intelligent Tiering for Gold to control storage cost. Compute is already decoupled and only on-demand . Storage we will cost control thru policies so whole solution cost is always managed and controlled.
- **SHIR** agent has no cost; only the VM incurs charges — we can automate and schedule the VM to run on-demand during ingestion windows to control cost.

Data Governance with Microsoft Purview



Microsoft Purview

- Data Governance is a key pillar to make **“data a product”** in a data mesh across the wider data fabric.
- Microsoft Purview provides a **centralized data catalog**, making assets discoverable across SAP, ADLS, Postgres, and BI tools.
- Enables **data search, classification, and scheduled discovery jobs**, helping teams quickly find and understand datasets.
- Offers **end-to-end lineage**, from SAP sources all the way to Tableau dashboards.
- Enforces **governance and security policies**, integrated with Azure AD for fine-grained access control.
- Ensures data is **auditable, compliant, and trusted** — critical for sustainability and regulatory reporting.
- **Data quality checks (Great Expectations)** make data trustworthy and AI-ready.
- TBH Purview is not important , point is “Data Governance” and its pillars are important to be implemented.Comparable to **Unity Catalog in Databricks** or **AWS Glue + Lake Formation in AWS**.

Risks & Mitigations

Risk	Mitigation
Schema drift in SAP tables	Great Expectations schema checks, alert & version control in Purview
Duplicate or missing CDC events	Use ODP delta tokens + idempotent MERGE logic in PySpark; Bronze append-only for replay
Data quality issues (nulls, referential breaks)	GE checks at Bronze/Silver, DLQ path with reprocess option
Hot partitions / skewed Spark jobs	Partition pruning, salt keys, Z-ORDER optimization
Spot node eviction on AKS	Use mixed pools (spot + on-demand), enable retries + checkpointing
Pipeline failures (ADF/Argo/Spark)	Central logging in Log Analytics; Argo retries; resume from Bronze
Performance bottlenecks (large tables)	Incremental CDC; push-down filters; scale-out Spark executors dynamically
Security & compliance gaps	Purview policies + Azure AD RBAC; masking in Postgres/BI; audit logging enabled
Cost overruns	Use autoscaling, scale-to-zero, file compaction, OPTIMIZE/VACUUM scheduling (explained before as well)

Summary & Next Steps

Summary

- We replace **pyRFC** with **ADF SAP CDC (ODP)** for reliable, incremental extraction.
- Data flows through **Bronze** → **Silver** → **Gold**, processed by **PySpark on AKS** orchestrated via **Argo**.
- **DV2 in Postgres** provides auditability; **Star marts** enable fast BI in Tableau/Power BI.
- **Great Expectations** + **Purview** ensure data is trusted, governed, and compliant.
- **Monitoring (Azure Monitor, Grafana)** and **cost controls (spot, scale-to-zero)** make the platform robust and efficient.

Next Steps

- Pilot extraction of a high-value SAP object (e.g., sales orders) via ADF CDC → Bronze.
- Validate Silver merge logic and SCD2 (slowly changing dim.) handling with PySpark.
- Stand up DV2 model in Postgres and expose star schema for BI.
- Configure Purview scans + Great Expectations data quality checks.
- Expand to more SAP objects and scale pipelines under Argo workflows.

Appendix (Optional , If Time permits)

ADF SAP CDC Connector Deep Dive

References :

Architecture : <https://learn.microsoft.com/en-us/azure/data-factory/sap-change-data-capture-introduction-architecture>

Get more information about the SAP CDC implementation details , prerequisites and capabilities:

- [Prerequisites and setup](#)
- [Set up a self-hosted integration runtime](#)
- [Set up a linked service and source dataset](#)
- [Manage your solution](#)

Hands on beginner ADF SAP CDC POC implementation when prerequisites and infra is in place:

https://www.youtube.com/watch?v=kdzC8uB_mM0

Idempotency

Idempotency means if we rerun the same CDC batch multiple times, the Silver table always ends up correct. We achieve this with deduplication by PK + `_cdc_ts` and a deterministic MERGE — only newer records update, older ones are ignored. This makes the pipeline retry-safe and audit-proof.”

At the **ingestion level** (SAP → Bronze), ADF **already prevents duplicates** and ensures *at-least-once but replayable* delivery. This gives us **idempotency at ingestion**: if the pipeline restarts, ADF resumes from the last committed token, not from the beginning.

Below step is applied from Bronze to silver to gold layer DT PySpark Job.

Idempotency helps when pipeline files with partial data load. We just need to fix the issue and re-run the pipeline, no code change needed and reruns are safe.

Idempotent Merge (PySpark / Delta Lake) (pseudo code) (src is bronze and tgt is silver and so on..)

```
silver.merge(src, "tgt.id = src.id") \
    .whenMatchedUpdate(
        condition="src._cdc_ts > tgt._cdc_ts",
        set={"col1": "src.col1"}) \
    .whenNotMatchedInsert(values={"id": "src.id", "col1": "src.col1"}) \
    .execute()
```

Medallion Architecture

References :

Best and clean explanation : <https://www.databricks.com/glossary/medallion-architecture>

Azure Medallion architecture explanation :

<https://learn.microsoft.com/en-us/azure/databricks/lakehouse/medallion>