ENTERPRISE DATA ARCHITECT CASE STUDY: DATA PLATFORM TRANSFORMATION FOR AI

THE CHALLENGE (THE SITUATION & MANDATE)

Our multi-tenant data platform is currently blocked from achieving AI capabilities due to legacy architectural debt. Mandate is to provide strategic and technical suggestions to our team on how to execute the full transformation.

Problem	Explanation (The Pain)	Strategic Impact (CSO)
Variable Schema & Embedded Data	Our platform is **low-code/no-code**, meaning the **schema is customer-specific** for the same application. Data is highly nested, and **critical Master Data is embedded**.	**Cost & Scale:** Leads to massive duplication, requires complex, inefficient processing logic, and drives high cloud costs.
Monolithic Lock-in & Efficiency	The system is proprietary and inefficient, operating in batch and preventing migration to cost-effective **OSS Spark/Kubernetes**.	**Cost & Latency:** Blocks near real-time data flow, preventing cloud cost optimization.
Low Latency Block	We cannot consistently process data at the required **5-10 minute latency** needed for real-time AI features and conversational agents.	**Observability & AI:** Hinders proactive monitoring and makes predictive use cases impossible.

THE SOLUTION AREAS

- The Foundational Fix: Data Layering, Decoupling, and Cost
- Achieving Real-Time Performance and Scalability
- Open-Source Transition, Observability, and Metadata

THE FOUNDATIONAL FIX: DATA LAYERING, DECOUPLING, AND COST

ARCHITECTURAL SUGGESTION

Medallion Architecture

Bronze Layer (Raw/Landing)
- Immutable, append-only storage of raw data as ingested
- Preserves complete audit trail and enables replay capability
- Handles variable schemas via schema-on-read patterns

Silver Layer (Cleansed/Conformed)
- Schema enforcement and validation
- Master data extraction and normalization occurs here
- Change Data Capture (CDC) processing for incremental loads

Gold Layer (Semantic/Feature)
- Business-oriented, denormalized aggregates
- Optimized for consumption (Analytics + AI)
- Pre-computed features for ML models

SEPARATION STRATEGY FOR VARIABLE SCHEMA

Approach: Parameterized Schema Registry + Dynamic Parser

1.Schema Registry Implementation:

- Central registry mapping tenant_id → schema_version → field_mappings
- JSON/Avro schema definitions per customer variant
- Version control for schema evolution

2.Master Data Extraction Process:

Bronze → Schema Detection → Master Data Extractor (Parameterized) → Silver

Parameters:

- Tenant configuration (which fields = master data)
- Extraction rules (XPath/JSONPath for nested data)
- Normalization mappings (customer_specific_field → canonical_field)

3.Lossless Fidelity Guarantee:

- Maintain Bronze layer with complete raw data (permanent audit trail)
- Use hash-based validation checksums at each layer transition
- Implement reconciliation framework comparing record counts and key metrics

COST JUSTIFICATION

Immediate Savings from De-duplication:

- **Example:** If Customer entity is embedded in 10 transaction types, each with 500K daily records
- **Current State:** 5M redundant customer records stored daily
- **Post-Normalization:** ~50K unique customer records + 5M lightweight references
- **Storage Reduction:** ~99% for master data components
- **Compute Reduction:** Eliminate redundant transformations on duplicated data

Real-World Impact:

- 40-60% reduction in storage costs (S3/ADLS)
- 30-50% reduction in compute time (fewer data movements)
- Improved cache efficiency (deduplicated data fits in memory)

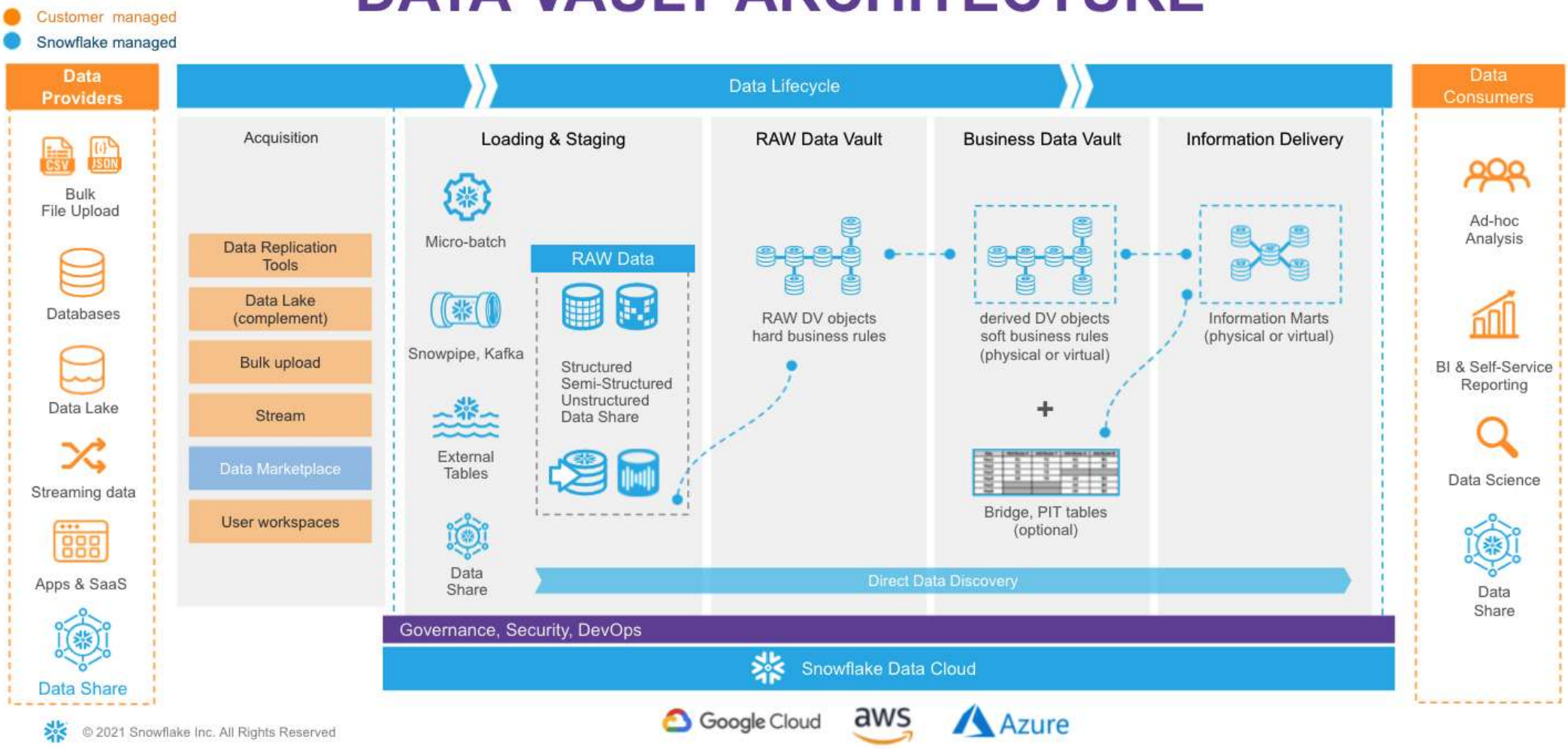
MODELLING PATTERN

Possible Solution: **Hybrid Data Vault + Feature Store**

Gold Layer Design:
For Analytics (Data Vault 2.0):
Hubs: Unique business keys (Customer, Product, Order)
Links: Relationships between entities (Customer-Order, Order-Product)
Satellites: Descriptive attributes with temporal tracking
Benefits: Handles schema changes, maintains history, supports multi-source integration

For AI (Feature Tables):
Entity-centric feature tables derived from Vault
Temporal features (30-day avg order value, customer lifetime metrics)
Real-time feature materialization for inference
Point-in-time correctness for training data

MULTI-TIER
DATA VAULT ARCHITECTURE




ACHIEVING REAL-TIME
PERFORMANCE AND
SCALABILITY

APPROACH: LAMBDA → KAPPA ARCHITECTURE EVOLUTION

Phase 1: Hybrid Lambda (Quick Win)

- Batch: OSS Spark for historical processing (Bronze → Silver)
- Speed: Flink/Kafka Streams for incremental updates (last 24-48 hours)
- Serving: Merge views in Gold layer

Phase 2: Pure Kappa (Target State)

- Unified stream processing (Flink on Kubernetes)
- Event-driven, continuous processing
- Achieves 5-10 minute end-to-end latency

CRITICAL POCS (PRIORITIZATION)

PoC #1: OSS Spark on Kubernetes + Iceberg/Delta Lake

- **Objective:** Validate monolithic processing logic migration
- **Success Criteria:**
  - Reproduce current batch outputs with 100% accuracy
  - Demonstrate 30%+ cost reduction vs. proprietary system
  - Prove Kubernetes autoscaling for variable workloads
- **Timeline:** 4-6 weeks
- **De-risks:** Vendor lock-in, compute costs, infrastructure modernization

PoC #2: Flink Streaming for High-Frequency Incremental Updates

- **Objective:** Achieve sub-30-minute latency for critical data flows
- **Success Criteria:**
  - Process 200K+ records/hour with <10 min end-to-end latency
  - Handle out-of-order events and late arrivals correctly
  - Demonstrate exactly-once processing semantics
- **Timeline:** 6-8 weeks
- **De-risks:** Real-time requirements, event ordering, state management

SCALABILITY STRATEGY

Horizontal Scalability:

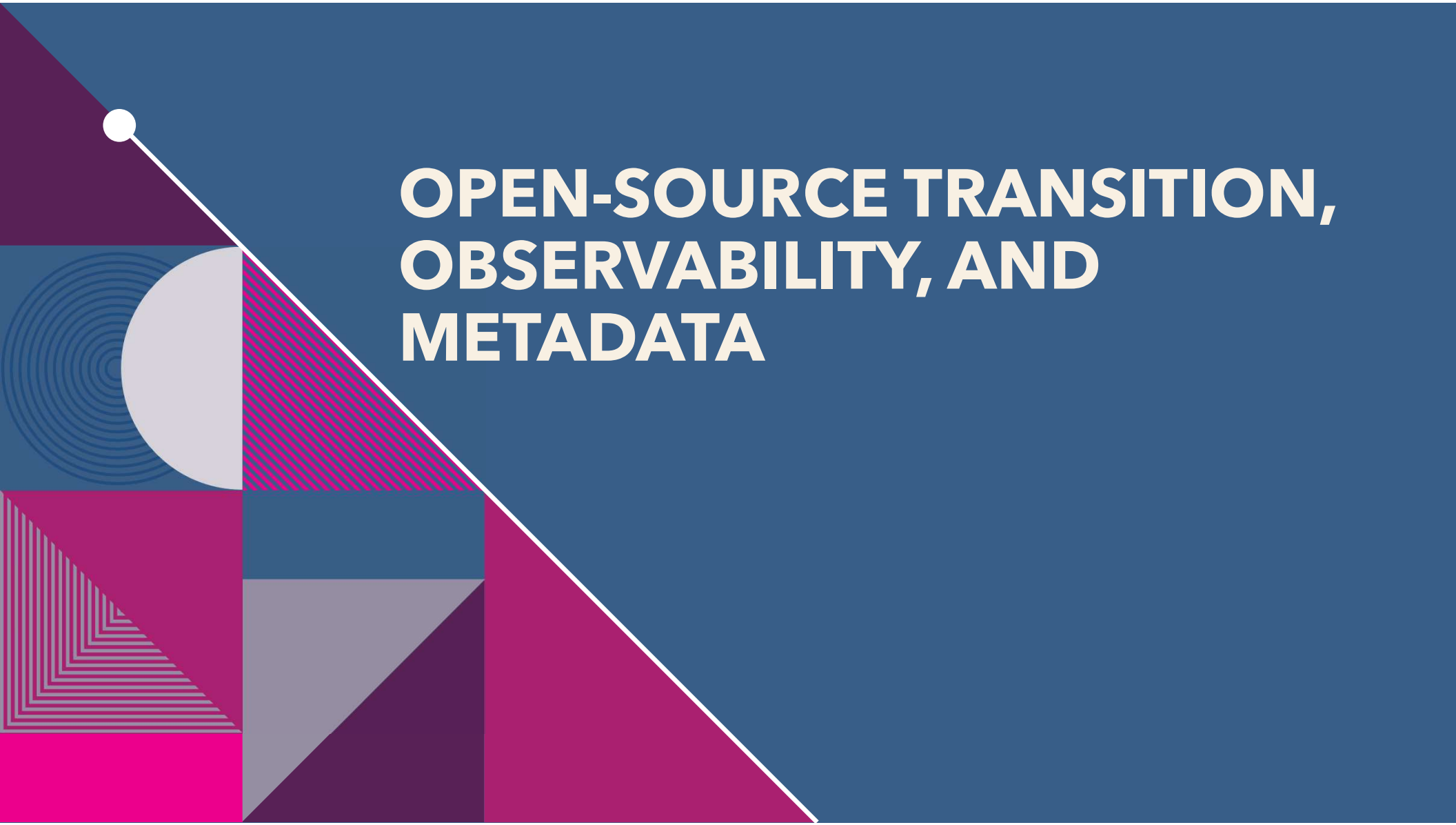
- **Partitioning:** By tenant_id and event_timestamp
- **Parallelism:** Dynamic partition count based on data volume
- **State Management:** Distributed state backend (RocksDB for Flink, Delta for Spark)

High Availability:

- **Checkpointing:** Regular state snapshots (every 5 minutes)
- **Multi-region replication** for critical Gold layer data
- **Circuit breakers** for dependent application data flows
- **Kafka as buffer** to decouple producers/consumers and handle backpressure

Handling Data Dependencies:

- **Watermarking strategy** for time-based joins across applications
- **Windowed joins** with configurable late-arrival tolerance
- **Fallback patterns** (serve stale data vs. block on missing dependencies)


OPEN-SOURCE TRANSITION,
OBSERVABILITY, AND
METADATA

RISK MITIGATION

Critical Migration PoCs

Already Covered in slide #12, Expanded Here:

PoC #1 - Extended Scope:

- **Logic Translation:** Convert proprietary ETL to PySpark/Scala
- **Performance Benchmarking:** Same input → same output, 1.5x faster target
- **Resource Optimization:** Right-size Kubernetes pods (CPU/memory profiling)

PoC #3 (Additional): Schema Evolution & Backward Compatibility

- **Objective:** Validate schema registry handles customer-specific changes
- **Test:** Add new field to 10 tenants, ensure no breaking changes to downstream
- **Success:** Zero-downtime deployment, automated schema migration

MANIFEST PATTERN DESIGN - OBSERVABILITY

Architecture:

Event Source → Manifest Creation → Processing → Manifest Update → Serving

Manifest Table Schema:

```
{
  manifest_id: UUID,
  tenant_id: String,
  source_system: String,
  batch_id: String,
  record_count: Long,
  start_time: Timestamp,
  end_time: Timestamp,
  status: Enum(RECEIVED, PROCESSING, COMPLETE, PAUSED, FAILED),
  quality_checks: JSON {
    schema_valid: Boolean,
    null_check_pass: Boolean,
    duplicate_count: Int
  },
  layer: Enum(BRONZE, SILVER, GOLD),
  lineage: Array[manifest_id] (parent manifests)
}
```

Purpose & Capabilities:

1.Granular Control:

- Pause/resume specific tenant data flows via status flag
- Replay from specific manifest_id for data corrections
- Rollback to previous manifest in case of issues

2.Enhanced Observability:

- Real-time dashboard showing processing status per tenant
- SLA monitoring (time in each layer)
- Quality metrics trend analysis
- End-to-end lineage tracking (source → Gold)

3.Operations Benefits:

- Root cause analysis (which manifest introduced bad data?)
- Capacity planning (processing time trends)
- Customer-specific SLA reporting

METADATA KNOWLEDGE BASE STRATEGY

Architecture: Centralized Data Catalog + Knowledge Graph

Components:

1.Technical Metadata (DataHub etc.)

- Schema definitions (fields, types, constraints)
- Physical location (table paths, partitions)
- Lineage (upstream/downstream dependencies)
- Statistics (row counts, null percentages, cardinality)

2.Business Metadata

- Non-technical names ("Customer Lifetime Value" vs. "cust_ltv_amt")
- Descriptions and calculation logic
- Data ownership and stewardship
- Sensitivity classification (PII, confidential)

3.Tenant-Specific Configuration

```
{
 tenant_id: "customer_A",
 application: "ERP",
 schema_version: "2.1.3",
 field_mappings: {
  "cust_ref_id" → "customer.external_id",
  "total_amt" → "order.total_amount"
 },
 join_conditions: {
  order_to_customer: "order.cust_ref_id = customer.external_id"
 }
}
```

Knowledge Graph Layer (Neo4j etc)

- **Entity relationships across applications**
- **Semantic search capabilities**
- **Traversal for impact analysis**

Critical for Conversational AI:

•**Natural Language Query Translation:**

User: "What's the average order value for premium customers this month?"

System:
1. Lookup "average order value" → metric_definition → calculation logic
2. Lookup "premium customers" → customer.segment = 'premium'
3. Lookup "this month" → time_filter → current_month()
4. Generate SQL from metadata + execute

•**Context-Aware Responses:**

- Understands tenant-specific field names
- Applies correct join paths based on schema version
- Surfaces data quality caveats from manifest metadata

STRATEGIC PIVOT: DATA FOR AI AND AGENT SERVICES

ARCHITECTURAL BRIDGE TO AI

How Low-Latency Foundation Enables AI:

Real-time Feature Freshness:

Sub-30-minute latency ensures model inputs reflect current state
Critical for fraud detection, dynamic pricing, recommendation engines

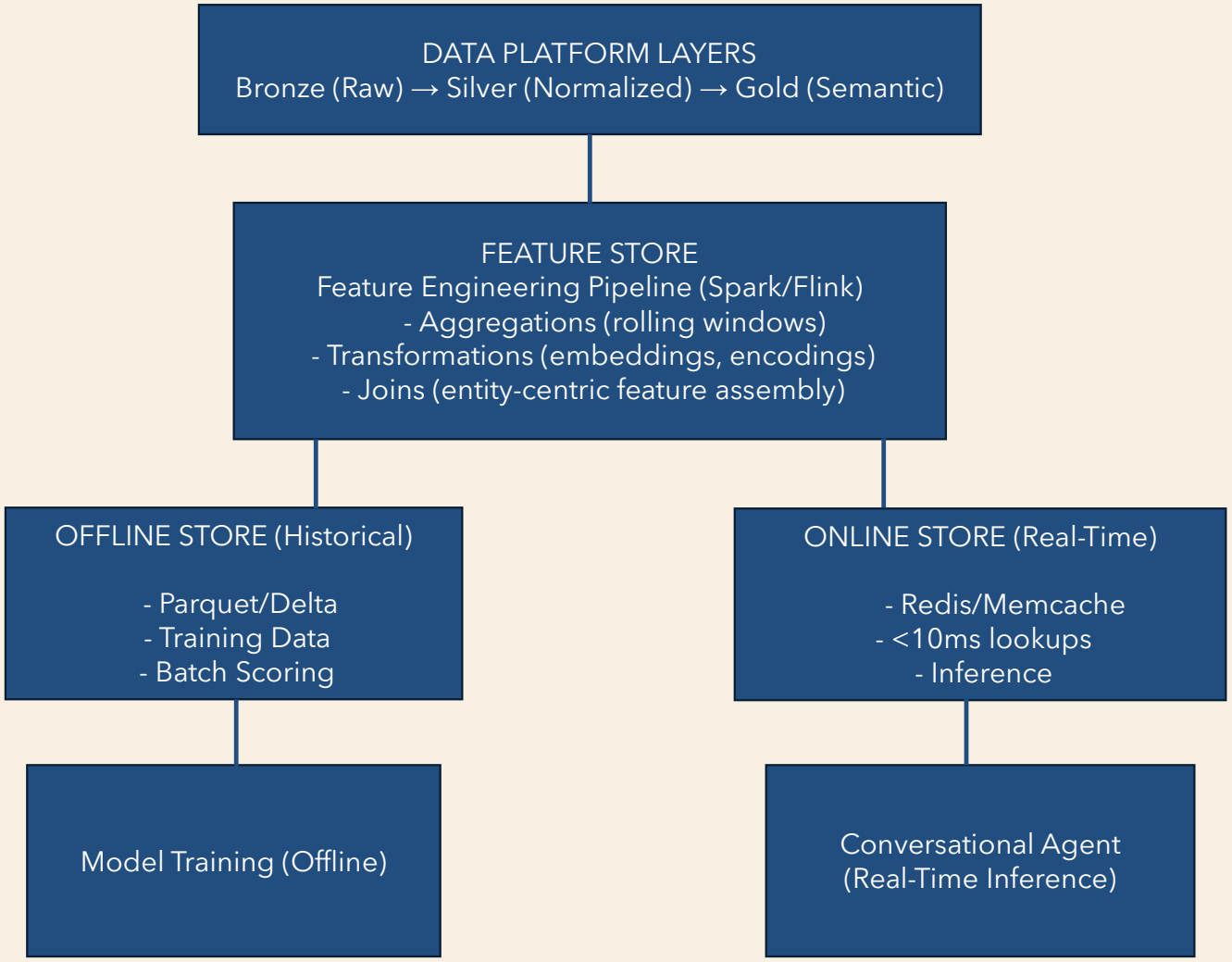
Normalized Data Access:

Master data separation enables consistent entity definitions
Reduces feature engineering complexity (no duplicate resolution logic)

Observability Infrastructure:

Manifest pattern provides training data lineage
Enables point-in-time feature reconstruction for model debugging

FEATURE STORE CONCEPTUAL FLOW



WHY FEATURE STORE IS NECESSARY

1.Dual-Timeline Consistency:
- Training: Point-in-time correct features (no data leakage)
- Inference: Fresh features with <100ms latency
- Challenge: Same feature definition, different serving requirements

2.Conversational Reporting Agents Needs:
- Low Latency: User asks question → response in <2 seconds
- Pre-computed Features: Can't run complex aggregations on-the-fly
- Contextual Awareness: Agent needs customer segments, behavioral features instantly

3.Reusability & Governance:
- Single source of truth for feature definitions
- Version control for features (reproducibility)
- Feature discovery (avoid duplicate engineering)

4.Example Use Case:
User to Agent: "How are my high-value customers trending?"

Agent Workflow:
1. Retrieve from Feature Store (online):
  - customer_segment features (cached)
  - 30_day_purchase_trend (pre-computed, refreshed every 30 min)
  - customer_lifetime_value (materialized)

2. Generate natural language response with visualization

Total latency: <1 second (vs. 5+ minutes if computing from scratch)

KEY DISCUSSION POINTS

CSO TRADE-OFFS: DATA QUALITY VS. LATENCY

Philosophy: "Quality gates should be asynchronous and non-blocking for real-time paths."

Implementation:

1.Multi-tier Quality Framework:

- **Tier 1 (Blocking):** Critical schema validation, null checks on primary keys → Silver layer entry gate
- **Tier 2 (Warning):** Statistical anomalies, referential integrity → logged in manifest, data flows through
- **Tier 3 (Async):** Deep quality rules, ML-based outlier detection → run post-processing, backfill corrections

2.Quarantine Pattern:

- Failed records routed to quarantine zone (S3/separate table)
- Main pipeline continues unblocked
- Operations team reviews and reprocesses quarantine batch

3.Progressive Quality:

- Bronze → Silver: Basic validation (5-10 min latency maintained)
- Silver → Gold: Richer quality checks asynchronously
- Gold layer: Highest quality guaranteed, but may lag by additional 10-15 minutes

Result: Achieve <30-minute latency for 99.5% of clean data, while maintaining high quality bar.

TEAM & LEADERSHIP STRUCTURE

Initial PoC Team (3-month sprint):

Core Roles:

1. **PoC Lead (Architect)** - Overall technical direction
2. **Data Engineer (2)** - PoC #1 execution
3. **Data Engineer (1-2)** - PoC #2 (Flink) execution
4. **Platform Expert (2)** - Domain knowledge, validation
5. **Data Engineer (1)** - Manifest pattern, observability
6. **DevOps/SRE (1)** - K8s infrastructure, CI/CD

Knowledge Transfer Strategy:

Phase 1 (Weeks 1-4): Learn & Shadow

- External consultants/OSS experts pair with existing team
- Daily standups focused on "why" not just "how"
- Hands-on workshops (Spark optimization, K8s debugging)

Phase 2 (Weeks 5-8): Collaborate & Build

- Existing team leads feature development with OSS stack
- Consultants provide code reviews and best practices
- Weekly architecture review sessions

Phase 3 (Weeks 9-12): Own & Extend

- Existing team runs PoC demos to stakeholders
- Begin onboarding additional team members
- Document patterns, runbooks, troubleshooting guides

Cultural Shift:

- Celebrate "unlearning" proprietary approaches
- Reward contributions to shared OSS knowledge base
- Hackathons focused on OSS ecosystem exploration

BUSINESS QUANTIFICATION: CLOUD COST SAVINGS

Conservative Estimate: 35-50% Total Platform Cost Reduction

Breakdown (Back of the envelope calculation):

1. Master Data De-duplication (Immediate):

- **Current State:** 5M records/day with 60% embedded redundancy
- **Post-Normalization:** 2M records/day
- **Storage Savings:** 60% reduction × $0.023/GB/month × 10TB = ~$8K/month
- **Compute Savings:** 40% fewer transformations = ~$15K/month
- **Subtotal:** ~$275K/year

2. OSS Migration (6-12 months):

- **Proprietary License Elimination:** ~$500K/year
- **Kubernetes Autoscaling:** 30-40% compute efficiency vs. fixed VMs = ~$200K/year
- **Spot Instances for Batch:** Additional 50-70% savings on batch compute = ~$150K/year
- **Subtotal:** ~$850K/year

3. Operational Efficiency (12-18 months):

- **Reduced Manual Interventions:** Better observability = 50% less ops time = ~$100K/year
- **Faster Troubleshooting:** Manifest pattern reduces MTTR = reduced downtime costs

Total Annual Savings: $1.2M - $1.5M (35-50% of estimated $3M annual platform cost)

Additional ROI:

- **Time-to-Market:** 3-6 months faster for new AI features = competitive advantage
- **Revenue Enablement:** Conversational AI could drive 5-10% upsell/cross-sell = $XM (business-dependent)

SUMMARY : KEY SUCCESS METRICS

Technical:

- Sub-30-minute end-to-end latency (95th percentile)
- 99.9% data quality score (post-Silver layer)
- Zero vendor lock-in (100% OSS stack)

Business:

- 40%+ cloud cost reduction
- Enable 3+ AI use cases in Year 1
- 50% reduction in schema change deployment time

Operational:

- <5 minute MTTR for data flow issues (manifest pattern)
- 100% lineage coverage for compliance
- Self-service feature creation for data scientists

This architecture positions the platform as an AI-ready, cost-efficient, observable foundation for next-generation data products.


THANKS