Part 1: Architectural Evolution

1. The Core Platform Evolution Timeline (2016–2024)

- **2016 (Foundational Operational Launch)**: The o9 platform achieved full functional maturity, onboarding its first production clients. At its architectural core sat the **GraphCube (Live Server / LS)**, a proprietary in-memory columnar database with an embedded rule-processing engine. To execute complex forecasting algorithms, the platform integrated an **R Server**. This enabled configuration consultants to utilize mature open-source R libraries and write declarative code via o9's proprietary language, **IRPL**, supplemented by procedural R compute loops.
- **2018 (The Big Data Pivot - BDP 1.0)**: o9 contracted with a massive global retail client whose supply chain data requirements exceeded legacy processing limits by 20x to 50x. Because the GraphCube ran on a single monolithic Windows server instance, it could not natively parse datasets of this size. This constraint catalyzed the **Big Data Platform (BDP) 1.0**, embedding the Hadoop and Hive ecosystems into the architecture. Billion-row tables (e.g., daily store-item transaction logs) were stored as Hive files. Small, highly aggregated slices were streamed into the GraphCube engine to drive front-end UIs and execute supply chain solvers, while results were written directly back to the Hadoop Distributed File System (HDFS).
- **2019 (Distributed Computation Integration)**: As clients demanded the ability to run custom Python models directly on the platform, standalone Python servers proved incapable of scaling to match large retail use cases. To achieve distributed execution, o9 integrated **Apache Spark** into BDP 1.0, allowing custom user-defined functions (UDFs) to execute across a scalable cluster of compute nodes. Concurrently, **Integration 2.0** was developed to manage enterprise API data transmission and real-time data ingestion.
- **2024 (The Modern Cloud-Native Paradigm - BDP 2.0)**: o9 rolled out **Big Data Platform (BDP) 2.0**, representing a major structural shift. While BDP 1.0 remains supported for legacy installations, new implementations are exclusively deployed on BDP 2.0. Apache Spark remains the core computation framework for PySpark execution, but the monolithic Hadoop, Hive, and HDFS frameworks have been **completely replaced by Delta Lake** operating directly on top of cloud object storage.

---

2. The Decommissioning of Legacy HDP (Hadoop)

The transition from BDP 1.0 (Hortonworks Data Platform) to BDP 2.0 was driven by critical limitations in legacy Hadoop-based architectures:

- **Complex and Brittle Infrastructure**: The legacy architecture relied on a complex, fragile web of legacy open-source tools—including **HDFS, Apache Zookeeper, Apache Ambari, Apache YARN, and Apache Hive**—which required immense engineering management overhead.

- **Severe Performance Bottlenecks**: High-volume workloads suffered from severe processing and read/write performance bottlenecks occurring at both the raw storage and live executor levels.
- **Stagnating Support and High Costs**: Open-source support for traditional Hadoop/Hive stagnated (becoming obsolete post-Cloudera acquisition). This forced companies into expensive proprietary commercial licensing agreements and drove up the Total Cost of Ownership (TCO).
- **Rigid Scaling Constraints**: In HDP, scaling clusters up or down created severe downtime overhead and operational service outages due to a complete lack of dynamic, cloud-native orchestration. YARN was responsible for forming the cluster with combined compute and storage node capabilities, making independent scaling of storage and compute impossible.

---

3. The Architecture of BDP 2.0 (The Cloud-Native Stack)

To resolve the constraints of the legacy infrastructure, o9 standardized on a modular, cloud-native data platform architecture consisting of four core components:

- **Kubernetes (K8s)**: Operating as the baseline container orchestration engine, Kubernetes manages microservices deployment, horizontal/vertical pod scaling, self-healing, and infrastructure-as-code consistency.
- **Apache Spark**: Acts as the in-memory distributed processing engine designed for ultra-high-speed calculations on massive, large-scale datasets.
- **Delta Lake**: Serves as the modern replacement for legacy Hadoop/Hive. It is an open-source, highly optimized storage layer that integrates directly with cloud object storage. It provides full **ACID (Atomicity, Consistency, Isolation, Durability) transaction compliance** on top of cloud storage layers, preventing partial data corruption during pipeline failures and ensuring structural write consistency.
- **Apache Airflow**: Serves as the workflow orchestration engine capable of managing complex **Directed Acyclic Graphs (DAGs)** and scheduling programmatic pipelines. Under **Integration 3.0**, graphical ETL pipelines mapped in the o9 UI are automatically compiled into production-grade Airflow DAGs behind the scenes.

---

4. Key Infrastructure and Storage Inventions in BDP 2.0

By migrating from a physical HDP cluster to a cloud-native BDP 2.0 environment, o9 revolutionized the underlying data movement and storage path:

- **Elimination of Physical HDFS**: BDP 2.0 completely eliminates physical Hadoop clusters and HDFS, replacing them with standard, native cloud object storage (such as AWS S3, Azure ADLS, or GCP GCS).

- **Dedicated Tenant Isolation**: The platform provisions dedicated, secure cloud storage buckets per client tenant, enforcing strict data boundaries.
- **Direct In-Memory Streaming**: BDP 2.0 facilitates **direct in-memory streaming reads and writes**. This completely removes the legacy necessity to copy intermediate, transient files down to physical local server directories during ETL execution, dramatically accelerating data transfer speeds.
- **On-Demand Compute Autoscaling**: Rather than maintaining expensive "always-on" physical compute nodes, BDP 2.0 leverages Kubernetes-managed Spark clusters for fluid resource allocation. Worker nodes, driver pods, and individual executors are spun up on-demand as intense workloads dictate, and are **automatically terminated immediately upon task completion** to minimize infrastructure footprint and waste.

Part 2: Delta Lake & Storage

1. Core Architectural Capabilities of Delta Lake

Delta Lake serves as the transaction management and storage layer of BDP 2.0, sitting directly on top of cloud object storage (such as AWS S3, Azure ADLS, or GCP GCS). It addresses a fundamental limitation of standard Spark engines, which natively only support basic append/overwrite file operations.

- **Granular Record-Level ACID Transactions**: Delta Lake enforces full **ACID (Atomicity, Consistency, Isolation, Durability)** transaction guarantees. This ensures that if an ingestion or data transformation pipeline fails mid-execution, the entire transaction is rolled back, preventing partial data corruption and maintaining write consistency. It also enables highly granular, record-level updates and hard deletes (standard SQL `UPDATE` and `DELETE` commands) on top of cloud storage.
- **Automated Schema Enforcement**: This mechanism safeguards downstream systems by automatically rejecting unexpected schema drift, malformed payloads, or incompatible data modifications before they can be ingested into a physical table.
- **Time Travel & Historic Querying**: Delta Lake tracks transaction history through structured log files. This permits developers to query older historical snapshots of a table at a specific point in time or version number, facilitating rapid rollbacks and debugging.
- **Columnar Parquet Compression**: Under the hood, Delta Lake stores data in highly compressed, columnar Parquet formats. Columnar storage allows BDP 2.0 to skip irrelevant rows and prune partitions during read operations, delivering exponentially faster query performance than traditional flat files (CSV/DSV).
- **The Schema Catalog**: Schema definitions, metadata characteristics, and table structures are registered centrally inside the **Delta Metastore** (leveraging the Hive Metastore Database), which is actively referenced by both Spark execution tasks and the Live Server to compile query pathways.

---

2. Syntax Invariants and Structural Data Organization

Migrating to a Delta Lake foundation introduces strict query rules and alters how the underlying data is organized:

- **Mandatory Delta Signature**: When writing DDL queries to initialize physical data structures inside the platform, the traditional `CREATE TABLE` command explicitly requires a **Delta Lake signature** to declare the table type.
- **Decommissioning of Hive Bucketing**: Legacy Hive bucketing configurations (previously used in BDP 1.0/Hadoop clusters) are completely unsupported under BDP 2.0. Data organization and physical file layout are now managed exclusively through **partitioning**.

- **Query Performance Optimization**: Query performance is heavily enhanced through **data skipping** and **partition pruning**. When executing queries, Delta Lake reads the metadata statistics of Parquet files to instantly skip files that do not contain the target keys, avoiding expensive whole-table scans.

---

3. Table Engine Classifications inside BDP 2.0

Not all tables residing within the BDP 2.0 ecosystem share the same storage performance and transactional safety profiles. The platform classifies tables into three distinct engines:

1. **External Tables**: These reference unmanaged external data files residing directly on cloud storage. They are typically leveraged during the raw, initial file ingestion stages of an ETL pipeline before the data is sanitized.
2. **Non-Transactional Tables**: These represent standard, basic Spark tables. They lack transactional safety guards, indexing structures, and performance optimization parameters. Because of the high risk of data corruption, their use is heavily discouraged.
3. **Delta Tables (o9 Standard)**: The recommended platform default. They natively unlock the complete suite of Delta capabilities—including full ACID transaction safety, automated schema enforcement, historical version logging, time travel, and automated file-compaction optimizations.

---

4. Housekeeping and Log Retention (`VACUUM`)

Because Delta Lake logs every single insert, update, and delete operation to preserve transaction history, tables will naturally accumulate stale, unreferenced Parquet files over time. To prevent storage bloat and optimize read performance, o9 enforces a structured housekeeping protocol:

- **The 7-Day Log Retention Standard**: By default, the system maintains a rolling **7-day transaction log retention window** to support historical query rollbacks and debugging.
- **The `VACUUM` Utility**: To clean up files older than the log retention threshold, the engine executes scheduled **`VACUUM` instructions**. Running `VACUUM` permanently deletes unreferenced data files from the underlying cloud object storage, reclaiming storage space and keeping the table's directory clean.

Part 3: Platform Component Mapping

1. BDP 2.0 Core Platform Components Mapping

The Big Data Platform 2.0 (BDP 2.0) infrastructure is composed of multiple specialized cloud-native services and microservices that work in unison to manage container scheduling, in-memory processing, data streaming, and logging. Below is the granular mapping of these core components:

- **Kubernetes (K8s)**: Serves as the foundational container orchestration engine. It directly manages software deployment, horizontal and vertical pod scaling, self-healing, and microservices lifecycle isolation. Kubernetes provides the highly scalable baseline infrastructure that hosts all functional BDP systems.
- **Spark Thrift Server (STS)**: An interactive, always-on SQL interface running on top of Spark SQL that exposes standard JDBC and ODBC connectivity endpoints. The STS completely replaces legacy Apache Hive and is actively queried by the GraphCube Live Server (LS), Integration 3.0 nodes, and Edge Node interfaces.
- **Apache Livy**: A secure REST-based interface designed to submit programmatic execution jobs directly to the active Spark cluster. It acts as the standard gateway used to submit complex computation plugins and custom ETL batch runs.
- **Cluster Scaler / o9 Livy-Proxy**: o9's proprietary intelligent service proxy layer wrapped directly over standard Apache Livy endpoints. It pre-warms and scales the node infrastructure ahead of time to drastically reduce cluster spin-up latency during time-critical execution routines.
- **Spark History Server (SHS)**: A web-based telemetry user interface that provides detailed, post-execution visual analysis of completed job blocks. Engineers use the SHS to inspect historical Spark jobs, step metrics, resource allocation charts, and granular execution logs.
- **HDAgent**: A purpose-built, background o9 system service that coordinates structured data movement and synchronization between the in-memory GraphCube server VM and the cloud-based Delta Lake storage bucket. It operates as the core data loader during active plugin execution flows.
- **DSN (Data Source Name)**: A native ODBC layer configured directly on the o9 Live Server instance. It establishes the physical connection pipeline required to pull or push data payloads directly to and from Delta Lake structures.
- **Vector (Log Forwarder)**: An ultra-lightweight, high-speed system log collecting and forwarding utility agent. It captures raw pod `stdout`/`stderr` logs and streams them to the cloud storage bucket so they can be parsed and visualized in the Spark History Server.
- **Volcano Scheduler**: A specialized, Kubernetes-native batch scheduling engine optimized for high-performance big data operations. It manages sophisticated scheduling queue logic, resource fair-sharing, and pod gang-scheduling for distributed Spark tasks within the cluster.

- **Delta Metastore (Hive Metastore)**: An open-source centralized metadata management catalog that tracks Delta Lake table schemas, partition indexes, and formats. It is actively referenced by both Spark and Delta processing layers to compile query pathways.

---

2. Kubernetes Pool Isolation Architecture (Shared vs. Isolated)

To prevent resource contention and guarantee security in multi-tenant environments, BDP 2.0 enforces a strict node segregation model:

- **Baseline Shared Services Pool**: Steady-state, shared platform services—such as the Spark Thrift Server (STS)—run on a **common, shared Kubernetes node pool** across the organization. This optimizes baseline infrastructure utilization for ongoing query exploration and reporting.
- **Dedicated Computation Isolation Pools**: Specialized, heavy-weight execution workloads (like Python/PySpark Plugins and Int3 ETL batch runs) are dynamically dispatched to **client-specific, isolated compute node pools**.
- **Pod Security and Separation Rules**: Kubernetes enforces strict tenant and organizational isolation within the cluster using **labels, node selectors, taints, and tolerations**. This ensures that one tenant's heavy computation tasks can never compromise the processing memory or stability of another tenant's environment.

---

3. BDP 2.0 Network Port & Routing Matrix

Traffic within the Kubernetes cluster and external service communication endpoints are routed through a highly disciplined matrix of dedicated ports:

- **Spark Thrift Server (STS)**: Kubernetes Port = `10000` | service Port = `31014`
- **Apache Livy REST service**: Kubernetes Port = `8998` | service Port = `31015`
- **STS Metastore (Hive Metastore Database)**: service Port = `9083`
- **Cluster Management Proxy (o9 Cluster Scaler)**: service Port = `31012`
- **Spark History Server (SHS) Telemetry UI**: service Port = `31016`
- **Hue UI (Delta Lake Data Explorer & File Browser)**: service Port = `31027`
- **HDAgent Local Connection**: Bound to Port `8087`. The GraphCube service routes connection properties through this port to establish physical streaming pipelines with the local HDAgent.

---

4. The Edge Node Concept

The **Edge Node** is an optional, standalone Virtual Machine instance deployed within the architecture to execute specific administrative and administrative-level integration workloads.

- **System Capabilities**: It acts as a secure gateway that hosts **Apache Beeline** (for direct SQL query exploration across Delta tables), **Apache Airflow** (for backend ETL DAG orchestration), and CLI utilities for programmatic ingestion and fast cloud-bucket-to-bucket synchronization.
- **Deployment Invariant**: The Edge Node is **completely optional**. If a client tenant environment relies exclusively on standard o9 Python compute plugins and native Integration 3.0 pipeline configurations, a physical Edge Node VM is not provisioned.

Part 4: Tenant Config Payloads & Routing Matrix

1. The Core Port Routing & Network Matrix

In a Big Data Platform 2.0 (BDP 2.0) cloud-native architecture, communication within the Kubernetes (K8s) cluster and external service endpoints are routed through a highly disciplined matrix of dedicated ports. This ensures traffic is cleanly isolated between administration utilities, interactive querying, and the core planning engine:

- **Spark Thrift Server (STS)**:
  - **Kubernetes (K8s) Port**: `10000`
  - **Service (Svc) Port**: `31014`
  - *Usage*: Exposes the interactive SQL execution layer for JDBC and ODBC incoming connections.
- **Apache Livy**:
  - **Kubernetes (K8s) Port**: `8998`
  - **Service (Svc) Port**: `31015`
  - *Usage*: Serves as the secure REST gateway for submitting Python or PySpark plugins and custom ETL workloads.
- **STS Metastore**:
  - **Service (Svc) Port**: `9083`
  - *Usage*: Coordinates schema validation, column mappings, and metadata lookups.
- **Cluster Management Port (Proxy)**:
  - **Service (Svc) Port**: `31012`
  - *Usage*: The gateway to the o9 Cluster Scaler proxy, which pre-warms nodes to minimize execution latency.
- **Spark History Server (SHS) UI**:
  - **Service (Svc) Port**: `31016`
  - *Usage*: Exposes the web interface used by data engineers to inspect historical job runs, resource allocation charts, and execution logs.
- **Hue UI**:
  - **Service (Svc) Port**: `31027`
  - *Usage*: Hosts the Delta Lake Data Explorer and o9 Data Lake File Browser.
- **HDAgent (Local Connection)**:
  - **Dedicated Port**: `8087`
  - *Usage*: The port through which the GraphCube Live Server communicates with the local background HDAgent to stream memory arrays down to the cloud bucket.

---

2. Tenant Configuration JSON Payloads

To successfully provision and route workloads within a BDP 2.0 tenant, administrators must configure specific JSON payloads inside the **Tenant System Settings**. These payloads control the connection mapping to Delta Lake, executor resource profiles, and local agent bindings.

A. `ExternalDataSourceConnections` Payload

This setting maps the connection properties that allow the `GraphCubeService` to establish ODBC DSN connections to target Delta Lake environments.

Raw JSON Payload Structure:

```
 [
  {
    "ConnectionName": "Deltalake Connection",
    "Description": "Deltalake Connection",
    "DataSource Type": "DeltaLake",
    "HDFSRootDir": "s3a://aps-awsgql0113",
    "ConnectionStringJson": "DSN=awsgq10\\13_delta",
    "Is Active": "true"
  }
]
```

-

B. `PyExecConfig` Payload

This setting defines how standard Python and PySpark plugins connect to Apache Livy, establishing the exact resource allocation constraints (CPU, memory, and nodes) for the temporary Kubernetes driver and executor pods.

Raw JSON Payload Structure:

```
 {
  "ExecutionMode": "Spark",
  "SparkConfig": {
    "BaseUrl": "<url>",
    "DriverMemory": "1G",
    "NumExecutors": 1,
    "DriverCores": 1,
    "ExecutorMemory": "1G",
    "MaxExecutorMemory": "45G",
    "MaxDriverMemory": "45G",
    "SparkEnvRootDir": "s3a://aps-awsgql0113/o9spark",
    "SparkEnvNfsDir": "H:/",
    "NumNodes": 3,
    "NumCores": 12
  }
```

}

-

C. `HdAgentConfig` Payload

This registers the binding address used by the `GraphCubeService` to locate and communicate with the local `HDAgent` utility over the designated port.

Raw JSON Payload Structure:
```
 {
  "AgentUrl": "http://localhost:8087"
}
```

-

D. `LivyBaseUri` Parameters

- **Functional Purpose**: This setting is used to explicitly override the `BaseUrl` defined inside the global `PyExecConfig` payload.
- **Operational Requirement**: It is strictly mandatory in **hybrid deployments** when more than one cluster is connected to a shared organization environment. Configuring `LivyBaseUri` allows the system to override the default Livy service configuration at the tenant level without affecting sister tenants.

---

3. Routing Dynamics in Hybrid Coexistence Deployments

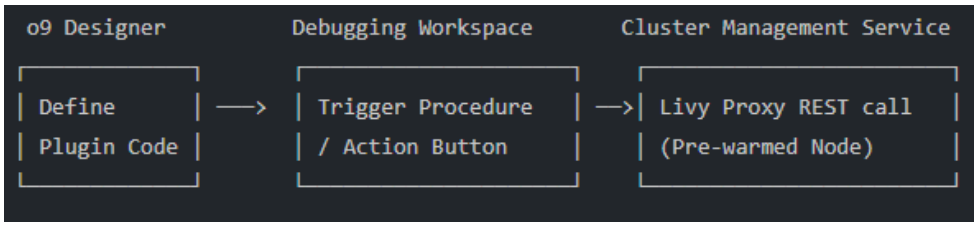
In enterprise environments transitioning to BDP 2.0, legacy HDP (Hadoop) clusters and modern BDP 2.0 environments commonly coexist. Because multiple tenants might share the backend infrastructure, o9 manages routing dynamically:

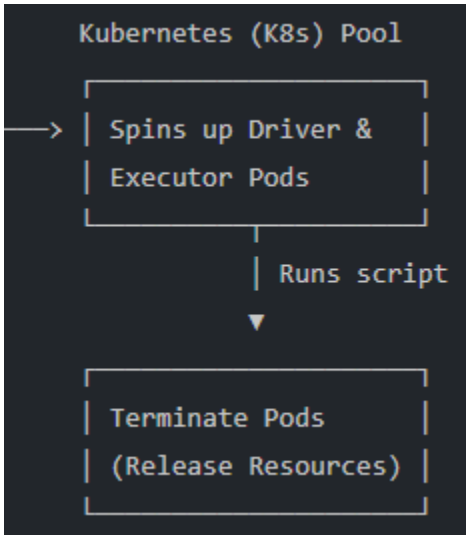
- **Tenant-Level Segregation**: For tenants migrated to BDP 2.0, parameters like `ExternalDataSourceConnections`, `PyExecConfig`, `HdAgentConfig`, and `LivyBaseUri` must be explicitly overridden at the individual tenant level.
- **HDP Fallback**: Tenants that have not yet completed their migration path will continue to default to the organization's legacy HDP connection pathways, preventing service disruption during multi-phase rollouts.

Part 5: Core Workload Exec & Sync Flows

1. The In-Memory Plugin Execution Lifecycle (Python & PySpark)

In BDP 2.0, custom computation and analytical modeling are executed through highly structured workflows using either **Python plugins** or **PySpark plugins**. These plugins run in an isolated, ephemeral containerized ecosystem rather than directly on the core Live Server VM.





Phase A: Definition and Registration

1. **Creation**: Configuration developers define the custom logic within the o9 Designer workspace.
2. **Constructor Blocks**:
    - Standard Python scripts use the `PythonScript` constructor block.
    - Distributed PySpark scripts use the `PySparkScript` constructor block.
3. **Physical Storage Path**: Plugins are saved under the global path: `Designer > Rule > Plugins > Global > o9.GraphCube.Plugins`.

Phase B: Invocation and Gateway Handshake

1. **Triggering**: The execution is initiated on-demand by an administrator inside the **Debugging Workspace** (via `Procedure > Action`) or systematically via scheduled batch execution processes.

2. **API Handoff**: The GraphCube engine connects to the **o9 Livy-Proxy** to pass the compiled code package.
3. **Proxy Routing**: The Livy-Proxy routes the execution payload to the active Apache Livy service via secure REST APIs while continuously monitoring performance telemetry.

Phase C: Kubernetes Pod Orchestration & Compute

1. **Container Spin-Up**: Apache Livy submits the job to the Kubernetes (K8s) scheduler. The cluster immediately provisions a dedicated **Driver Pod** and dynamic **Executor Pods** within the client tenant's isolated compute node pool.
2. **Pre-Warming**: To eliminate cluster spin-up latency during interactive planning tasks, the **o9 Cluster Scaler** proxy pre-warms compute nodes fractions of a second before the task is scheduled to run.
3. **Execution and Immediate Teardown**: The script runs entirely in-memory inside the temporary pods, processes the data, and writes the output back. To minimize infrastructure waste and cost, **all driver and executor pods are automatically destroyed immediately upon task completion**.

---

2. The Telemetry and Observability Pipe

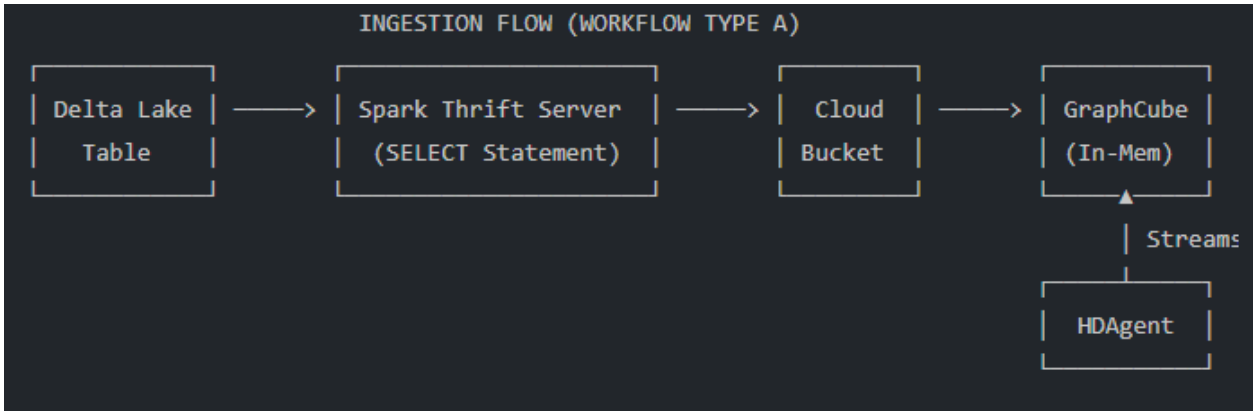
Because execution pods are ephemeral and deleted upon completion, logging must be captured in real-time to prevent log loss. BDP 2.0 utilizes an automated logging pipeline to enable deep post-execution debugging:

1. **Log Capture**: While the Spark/Python pods are running, the ultra-lightweight **Vector** log forwarding agent captures raw `stdout` and `stderr` streams directly from the containers.
2. **Cloud Storage Ingestion**: Vector streams these log lines in real-time out to the tenant's dedicated cloud object storage bucket.
3. **Visualization Gateway**: System developers can open the **Spark History Server (SHS)** or the **Cluster Jobs View** inside the Kibo Debugging Workspace to visually parse the logs, trace driver/executor memory consumption, and review Spark query DAG paths.

---

3. External Data Synchronization (Sync Local & Sync External)

To move data between the in-memory **GraphCube (Live Server)** VM and the **Delta Lake** tables sitting on cloud storage, the platform defines two specialized synchronization data paths. These operations are executed via database procedure scripts written in **IBPL**.



Workflow Type A: External Table to GraphCube Ingestion (Sync Local)

This flow is used to pull processed data from Delta Lake tables to hydrate the active in-memory planning matrices of the GraphCube.

1. An IBPL sync script issues an SQL `SELECT` instruction targeted at an external Delta table.
2. The **Spark Thrift Server (STS)** processes the SQL query, retrieves the data from Delta Lake, and writes a temporary compiled file out to the tenant's cloud bucket.
3. The **HDAgent** (running on the Live Server VM) establishes a local streaming port (Port `8087`) and pulls the file down from the cloud storage bucket.
4. The **GraphCube** engine parses the incoming file stream and instantly populates its local, volatile in-memory columns.

Workflow Type B: GraphCube to External Table Export (Sync External)

This flow is used to push updated planning measures from the in-memory GraphCube back into the persistent Delta Lake storage layer.

1. The GraphCube engine compiles its active memory matrix into an outbound stream.
2. The local **HDAgent** processes this stream and pushes the structured dataset directly to the tenant's cloud storage bucket via secure, encrypted ODBC DSN connection pathways.
3. The data is written and serialized into high-performance, transactional **Delta Lake tables** on the cloud bucket, ready for downstream consumption.

---

4. Supply Chain Solver (SCS) Execution Pattern

Standard planning operations require writing all results to the Live Server's memory [15). However, because o9's Supply Chain Solvers (SCS) handle extremely heavy data matrices that would exhaust standard RAM, BDP 2.0 implements a specialized execution bypass:

- **Bypassing Live Server Persistence**: When executing a solver run, the SCS engine completely bypasses persistent storage within the Live Server (LS) VM.
- **Direct Cloud Streaming**: Instead, the solver interacts directly with Delta Lake using **direct streaming reads and writes** straight to and from the assigned cloud storage bucket locations.
- **Execution Mechanics**: The user defines the explicit *External Data Configuration* inside the workspace UI, deploys the schemas, and triggers the solver plugin via an IBPL database query. The solver pulls the raw data from the cloud bucket, processes the calculations within the K8s Spark cluster, and streams the optimized outputs directly back to Delta Lake, protecting the active Live Server memory from crash-inducing resource spikes.

Part 6: Resource mgmt

1. Compute Node Pools & Workspace Isolation

To guarantee resource optimization and security in multi-tenant environments, BDP 2.0 segregates its processing nodes into two types of Kubernetes (K8s) node pools:

- **Baseline Shared Services Pool**: Steady-state, continuous platform services—such as the **Spark Thrift Server (STS)**—run on a **common, shared Kubernetes node pool** across the organization. This optimizes baseline infrastructure cost and supports ongoing lightweight querying.
- **Dedicated Computation Isolation Pools**: High-performance, isolated computation workloads (such as Python/PySpark plugins and Integration 3.0 ETL batch runs) are dynamically dispatched to **client-specific, isolated compute node pools**.
- **Pool Constraints**: Compute node pools are configured with explicit minimum and maximum node thresholds alongside highly granular CPU and RAM allocation boundaries.

---

2. Autoscaling Mechanics & Pre-Warming

Unlike legacy, always-on physical Hadoop clusters, BDP 2.0 scales cluster resources fluidly on-demand to handle varying workloads:

- **Fluid Pod Scaling**: The Cluster Management Service (CMS) dynamically scales the Kubernetes cluster footprint up and down. When a PySpark script or ETL pipeline is invoked, dynamic executors are spun up based on the resources required. Upon job completion, these pods are **automatically terminated and destroyed** to release resources and eliminate idle compute costs.
- **The Pre-Warming Engine**: To counteract the natural spin-up latency of provisioning new Kubernetes nodes, the **Cluster Scaler (Livy Proxy)** is designed to **pre-warm compute nodes** fractions of a second before time-critical, interactive planning tasks execute, dropping cluster spin-up latency to zero.

---

3. Spark ParamSet Tuning (Preventing OOM Crashes)

To tailor the resource consumption of individual scripts and prevent container crashes, developers specify runtime arguments via **Spark ParamSets** inside the Integration workspace.

Below is the definitive reference matrix of parameters utilized to manage resources, optimize query plans, and prevent Out of Memory (OOM) failures:

Parameter	Operational Impact & Usage
`spark.driver.memory`	Allocates RAM to the Spark Driver Pod (e.g., `2g`). Governs the volume of data the driver can aggregate.
`spark.driver.memoryOverheadFactor`	Defines the overhead memory safety margin (e.g., `0.3`) to prevent JVM container-level OOM crashes.
`spark.executor.memory`	Allocates RAM per individual Executor Pod (e.g., `2g`). Determines per-executor processing capacity.
`spark.executor.cores`	Controls parallel task loops inside each executor (typically set to `2` to balance thread capacity).
`spark.executor.instances`	Controls the static count of executor nodes when dynamic allocation is bypassed.
`spark.dynamicAllocation.enabled`	Enables automated cluster autoscaling (`true`) to dynamically add or remove executors at runtime.
`spark.dynamicAllocation.minExecutors` / `maxExecutors`	Establishes the floor (e.g., `2`) and ceiling (e.g., `4`) boundaries for active executor counts.
`spark.sql.adaptive.enabled`	Activates **Adaptive Query Execution (AQE)** to optimize query plans at runtime based on intermediate data statistics.
`spark.sql.adaptive.coalescePartitions.enabled`	Dynamically coalesces small post-shuffle partitions to minimize shuffle bottlenecks.
`spark.sql.adaptive.skewJoin.enabled`	Automatically detects and optimizes skewed joins by splitting heavily skewed partitions into smaller sub-partitions.

`spark.sql.autoBroadcastJoinThreshold`	Determines the threshold size for executing an optimized **Broadcast Join**. Setting this parameter to `-1` explicitly disables broadcast joins to prevent driver memory OOMs on large lookup tables.
`spark.sql.broadcastTimeout`	Establishes the timeout ceiling (e.g., `1200s`) to protect heavy table broadcasts from failing prematurely.

---

4. Workload State Telemetry

Workloads managed via the CMS transit through four explicit job lifecycle states, which can be monitored inside the Kibo Debugging Workspace under the **Cluster Job Details** view:

1. `PENDING`: The job is queued and waiting for cluster resource allotment.
2. `RUNNING`: The job is actively executing processing threads on the Spark cluster.
3. `COMPLETED`: The task finished execution successfully and exited without errors.
4. `FAILED`: The processing thread encountered a fatal exception and terminated abnormally.

---

5. Troubleshooting & Diagnostics Matrix

When execution failures occur, administrators use the following diagnostic pathways to isolate and resolve the root cause:

A. Jobs Stuck in `PENDING`

- **Symptom**: A PySpark plugin or ETL pipeline remains in a `PENDING` state for an extended period.
- **Root Cause**: **Node Pool Starvation**. The workload's resource requests (driver/executor memory or CPU cores) exceed the maximum auto-scaling capacity defined for the cluster's node pool.
- **Resolution Options**:
    1. Inspect the Kubernetes scheduler event log inside the **Cluster Jobs View** to confirm capacity blocks.
    2. Submit an administrative request to scale out and increase the maximum node pool limits.

3. Sequence the execution of batch jobs, or reduce the resource allocation parameters inside the active Spark ParamSet to fit within the existing node bounds.

B. Out of Memory (OOM) Crashes

- **Symptom**: Pod executors are repeatedly killed and recreated mid-job, leading to a `FAILED` state.
- **Root Cause**: The volume of data processed exceeds the RAM allocated to individual Spark executor pods.
- **Resolution**: Edit the Spark ParamSet and increase the **`spark.executor.memory`** threshold. If driver-level aggregation crashes occur, increase **`spark.driver.memory`** and **`spark.driver.memoryOverheadFactor`**.

C. Missing Conda Environments

- **Symptom**: A Python or PySpark script fails immediately upon invocation with an unhandled package import exception.
- **Root Cause**: The custom Python script references standard or third-party libraries (e.g., specific pandas/numpy versions) that have not been provisioned inside the running pod's active Conda template.
- **Resolution**: Navigate to the **Python Environment** tab in the Deployment Workspace, verify that the required Conda template (e.g., template version `2023_11`) is successfully provisioned and active, and click **Apply Changes and Reset Env** to re-sync the running containers.

D. Configuration Mismatches

- **Symptom**: The pipeline fails with immediate connection errors or invalid directory paths.
- **Root Cause**: Tenant-level settings are out of sync. The tenant is pointing to obsolete, legacy HDP pathways or invalid, hardcoded local HDFS syntax instead of cloud-native storage targets.
- **Resolution**: Audit the **Tenant System Settings**. Ensure that `ExternalDataSourceConnections` maps DSN targets cleanly to Delta Lake, and that the `HdAgentConfig` is explicitly routed through the designated local background port **`8087`**.

E. External Schema Drift

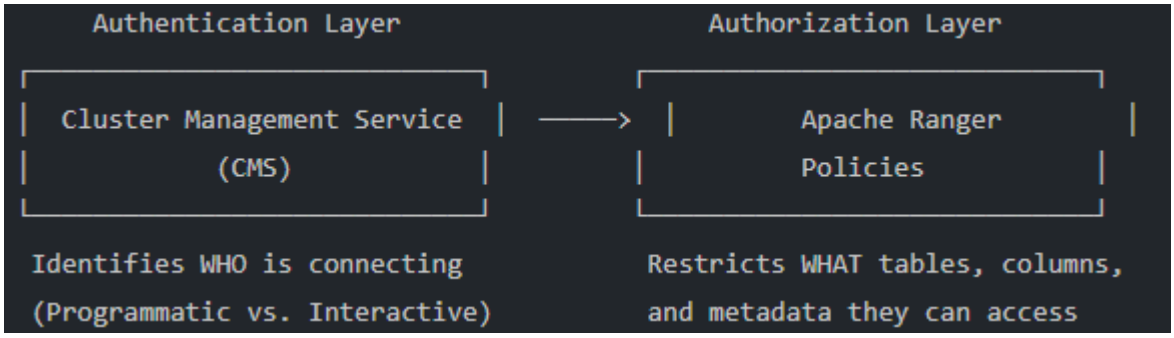
- **Symptom**: Data writes fail with critical validation exceptions during Delta Lake table ingestion.
- **Root Cause**: The structural data column types arriving from raw external source tables mismatch the metadata definitions registered in the central **Delta Metastore (Hive Metastore)**.

- **Resolution**: Run a direct SQL query against the table schema inside the **Delta Lake Data Explorer (Hue UI)** to inspect column characteristics. Adjust the incoming raw schema or update the Target Table mappings inside the EKG Configurator to align data types.

Part 7: BDP Security

1. The Two-Tier Security Architecture

To protect sensitive enterprise data while facilitating high-speed, distributed Big Data execution, BDP 2.0 implements a strict **two-tier security boundary**:



- **The Authentication Layer (The Gateway)**: Managed directly by the **Cluster Management Service (CMS)**. The CMS acts as the identity provider, verifying the credentials of any user, pipeline, or external system attempting to access the Kubernetes cluster.
- **The Authorization Layer (The Guard)**: Governed by **Apache Ranger** policies. Apache Ranger enforces fine-grained data access constraints, controlling which authenticated identities can interact with specific Delta Lake tables, execution queues, or schemas.

---

2. System Security Access Levels

BDP 2.0 support parameters configured at the system level by **ThriftOps** to dictate the strictness of the security posture. The platform operates under three distinct **System Security Access Levels**:

1. `None`: Security checking rules are entirely disabled. This level is strictly restricted to sandboxes, local training environments, or isolated local deployments.
2. `Auth`: Standard authentication verification protocols are enforced. The system validates identity credentials but relies on basic, non-Ranger database permissions for data-level access.
3. `Full`: The maximum security standard. Both strict end-to-end user authentication and **Apache Ranger data authorization** checks are actively enforced. Every single query and execution task undergoes immediate, fine-grained access audits.

---

3. Operational BDP Roles

Authorization within the BDP 2.0 workspace segregates users and system service accounts into three core operational profiles:

- **`BDP Admin`**: Grants **unrestricted root-level execution** and structural modification access across all tables, schemas, and Kubernetes clusters. BDP Admins can define schemas, provision node pools, and manage global settings.
- **`BDP User`**: Permits standard read and write query execution profiles. This is the default role for data engineers and integration pipelines, allowing them to ingest data, execute PySpark scripts, and write results back to Delta Lake.
- **`BDP Reader`**: Restricts the identity to a **read-only (`SELECT` only)** query execution profile. BDP Readers can query and explore data tables but are blocked from inserting, updating, deleting, or altering table schemas.

*Note: Access control profiles and tenant roles are managed and updated dynamically via the platform's native **Role Management UI**.*

---

4. Connection & Gateway Authentication Protocols

BDP 2.0 defines two discrete pathways to validate connection requests depending on whether the traffic is a systematic, background ingestion run or an ad-hoc, manual user query:

- **Programmatic / System Ingestion**: When automated pipelines, background sync procedures, or the Live Server (`GraphCubeService`) attempt to write to or read from Delta Lake, they bypass standard login challenges. Instead, the system authenticates the connection programmatically using the secure, internal **`BDPAuthSecretToken`**.
- **Interactive User Accounts**: When configuration engineers, data stewards, or administrators attempt to run ad-hoc SQL queries or test plugins, they authenticate using unique, **user-generated API keys**. This allows the platform to tie every manual query to a physical user identity for compliance and auditing.

Part 8: GLPS on K8

Part 8: Gurobi Linear Programming Solver (LPS) on K8s

The Big Data Platform 2.0 (BDP 2.0) natively supports running high-throughput supply chain optimization workloads inside a containerized Kubernetes environment using the Gurobi Linear Programming Solver (LPS).

To deploy and execute these advanced optimization procedures safely and dynamically, the platform enforces strict licensing models, volume-mount configurations, and operational parameters.

---

1. Supported Gurobi Licensing Models

Traditional licensing structures—specifically **node-locked MAC-address bindings** or standard **Compute Server Licenses**—are **completely incompatible** with the ephemeral, shifting nature of a Kubernetes cluster. Instead, BDP 2.0 is officially certified to run Gurobi engine version `10.0.2` using two cloud-compatible licensing paths:

- **Web License Server (WLS)**: Requires both a client and a compute license. The compute license is mounted on a dedicated Compute Pod to authorize the solver's local execution threads.
- **Cloud License**: Requires only a client license file. Under this model, the containerized environment sends mathematical formulations directly to Gurobi's cloud service for solving, bypassing the need for a persistent local compute license.

---

2. Kubernetes Secrets & Volume Mount Paths

To protect intellectual property and secure access credentials, all Gurobi solver licenses are loaded and stored securely as **encrypted Kubernetes Secrets** managed by ThriftOps.

When an optimization script is invoked, the platform utilizes a specialized JSON volume-mount payload to dynamically map the client license secret straight to the ephemeral driver and executor pods.

Gurobi Solver Secrets Volume Mount Payload

```
{
  "spark.kubernetes.driver.secrets.gurobi-client-lic": "/opt/gurobi",
  "spark.kubernetes.executor.secrets.gurobi-client-lic": "/opt/gurobi"
}
```

- **WLS Licensing Requirement**: If using the Web License Server (WLS) model, the compute license file must be explicitly mounted at `/opt/gurobi/gurobi.lic` on the running Compute Pod.

---

3. Operational Execution Invariants

When executing optimization solver routines on the cluster, developers must configure specific execution variables to prevent compilation crashes:

- **Mandatory WLS Parameter**: When triggering optimization procedures via WLS, the exit plugin command must explicitly pass the dynamic parameter `LPSN_K8_mode=true`. This flag tells the execution compiler to resolve licensing across the Kubernetes cluster topology.
- **Dynamic Pod Orchestration**: Once the optimization procedure is fired, the platform scales dynamically. It launches a temporary Python container pod alongside a **dedicated Gurobi optimization solver pod** in Kubernetes, executes the solver loops, and tears down the pods as soon as the results are safely synchronized.

---

4. Technical Consultant Readiness Checklist

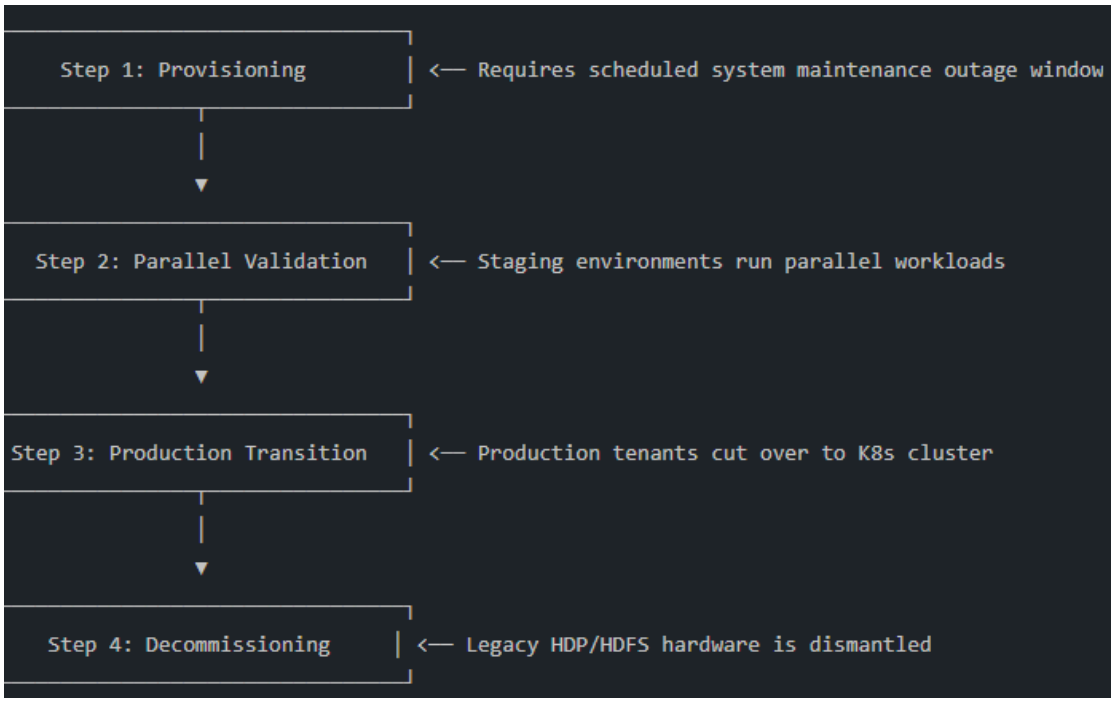
Before launching a production solver run on the BDP 2.0 cluster, implementation consultants must execute this **6-step readiness verification sequence**:

1. **Software Compatibility**: Confirm that all core platform software build versions are fully updated and certified compatible.
2. **System Parameters**: Verify that all tenant-level Delta Lake configuration system settings are correctly aligned.
3. **Solver Versioning**: Validate that the Gurobi engine version is exactly `10.0.2`.
4. **License Injection**: Verify with ThriftOps that the solver licenses have been successfully injected and registered inside the Kubernetes secrets database.
5. **The "Golden" Run**: Execute an initial validation test using a certified, known "Golden" engineering reference dataset.
6. **Package Alignment**: Verify that the running pods match the required Conda environment package layout.

Part 9: Migration & Coexistance

1. The 4-Step Migration Roadmap: Legacy HDP to BDP 2.0

Transitioning an enterprise environment from the legacy Hortonworks Data Platform (HDP/BDP 1.0) to Big Data Platform 2.0 (BDP 2.0) is executed through a highly disciplined, multi-phase lifecycle designed to guarantee zero data loss and minimize business disruption.



- **Step 1: Provisioning (Cluster Initialization)**:
  - **Action**: Spin up the new cloud-native BDP 2.0 Kubernetes (K8s) cluster, metadata databases, and associated microservices.
  - **Operational Invariant**: This milestone **requires an intentional, scheduled system maintenance outage window** to safely wire up the foundational network routing layers.
- **Step 2: Parallel Validation (Staging Verification)**:
  - **Action**: Connect non-production and staging environments to the newly provisioned BDP 2.0 cluster.
  - **Execution**: Run active data ingestion, ML plugins, and analytical pipelines on the new cluster in parallel with legacy runs.
  - **Validation**: Data engineers execute thorough validation routines, conducting cell-by-cell schema and output comparisons against the legacy HDP outputs to certify statistical and functional consistency.
- **Step 3: Production Transition (Cutover)**:
  - **Action**: Formally cut over the live production customer tenants to the active BDP 2.0 Kubernetes cluster.

    - **Mechanics**: Update tenant-level settings to point to the secure Delta Lake cloud storage buckets, direct ODBC DSN paths, and Livy proxy gateways.
- **Step 4: Decommissioning (Legacy Clean-up)**:
    - **Action**: Safely wipe all historical HDP tenant database instances and decommission the physical legacy HDP cluster infrastructure.
    - **Operational Invariant**: This step **requires a minor maintenance outage window** to prevent routing exceptions or orphan connections as legacy nodes are dismantled.

---

2. Hybrid Coexistence Deployment Architecture

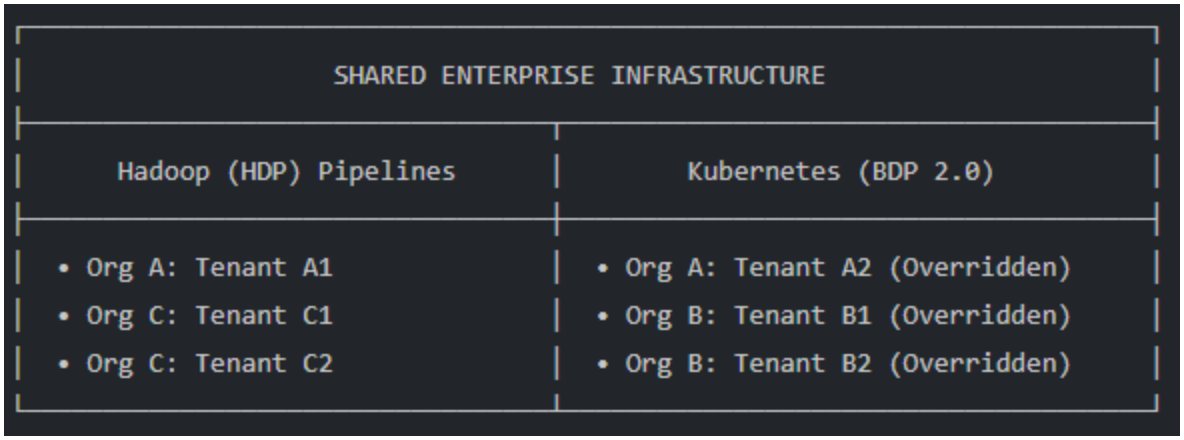
During large enterprise rollouts, a complete immediate transition of all business units is rarely feasible. Consequently, legacy HDP and modern BDP 2.0 platforms must frequently **coexist within the same shared backend environment**.

- **The Backend Shared Mandate**: If multiple tenants share a backend environment, the **HDP platform must remain fully active on the backend** until every single sister tenant completes its migration path.
- **Tenant-Level Segregation (Platform Overrides)**: To keep some tenants on HDP while others utilize BDP 2.0, administrators enforce isolation by overriding four critical configuration parameters strictly at the **individual tenant level**:
    1. `PyExecConfig`: Routes Spark executor/driver parameters to K8s instead of YARN.
    2. `ExternalDataSourceConnections`: Points the tenant to cloud-native Delta Lake instead of legacy Hive.
    3. `HdAgentConfig`: Connects the Live Server to the local background `HDAgent` utility over port 8087.
    4. `LivyBaseUri`: Overrides the global Livy service URL to route requests straight to the o9 Cluster Management Proxy.
- **Default Fallback**: Any tenant that does not have these explicit overrides configured will automatically default to the organization's legacy HDP connection and execution pathways.

---

3. Structural Multi-Tenant Isolation (Organizational Matrix)

In a hybrid coexistence deployment, o9 isolates distinct tenant workloads and platforms. Consider the following multi-tenant organizational matrix illustrating how resources are routed across shared infrastructures:



- **Organization A (ID: 2) — Hybrid Slicing**:
  - **Tenant A1**: Continues to operate on **legacy HDP**. It uses default system settings and YARN resource management.
  - **Tenant A2**: Migrated to **BDP 2.0**. It requires explicit tenant-level overrides for `PyExecConfig`, `ExternalDataSourceConnections`, `HdAgentConfig`, and `LivyBaseUri`.
- **Organization B (ID: 3) — Full BDP 2.0 Standard**:
  - **Tenant B1 & Tenant B2**: Both tenants run exclusively on **BDP 2.0**. Their workloads are routed to isolated Kubernetes compute node pools using tenant-specific parameters.
- **Organization C (ID: 4) — Legacy Holdout**:
  - **Tenant C1 & Tenant C2**: Both tenants run on **legacy HDP**. They require zero tenant-level overrides and continue executing workloads across standard HDFS and Hive structures.

---

4. Technical Retrofitting & Legacy Conversions

To finalize the migration to BDP 2.0, implementation teams must adapt legacy data pipeline technologies to the new cloud-native architecture:

- **Decommissioning legacy SSIS**: Legacy SQL Server Integration Services (SSIS) pipelines must be systematically redesigned and rewritten into low-code **Integration 3.0 pipeline configurations**. This transition shifts the processing logic out of physical database servers and onto distributed PySpark executors running on Delta Lake.
- **DAG and Script Retrofitting**: Custom external ETL scripts and external Apache Airflow DAG structures must be refactored to **interface directly with the modern Edge Node endpoints**. Hardcoded paths pointing to legacy local HDFS paths (e.g., referencing raw node directories) must be updated to target secure S3/ADLS/GCS cloud object storage base paths (`o9DataLakeBasePath`).