Department of Computer Science

Software Engineering Practice (CS 306)

DataBase Management System Project Report

OraLake – DataLake Using OracleDB

Implementation Documentation and Practical Guide

Submitted By

Siddharth Karmokar (123CS0061)
Kumar Bhaskar (123CS0056)

Supervisor

Dr. Srinivas Naik

Mrs. Kotipalli Padmasri

October 2025

Problem Statement

Title of the Project: Oralake – DataLake Using OracleDB

Name of the Student(s):
Siddharth Karmokar (123CS0061)
Kumar Bhaskar (123CS0056)

Examiner(s):

Supervisor(s):

Head of Department:

Date:

Declaration

We, **Siddharth Karmokar (123CS0061)** and **Kumar Bhaskar (123CS0056)**, hereby declare that the material presented in the Project Report titled **"Oralake – Data-Lake Using OracleDB"** represents our original work carried out in the **Department of Computer Science and Engineering** at the **Indian Institute of Information Technology, Design and Manufacturing, Kurnool** during the academic year **2025**. With our signatures, we certify that:

- No data, figures, or results have been manipulated or fabricated.
- No part of this report has been plagiarized from external sources.
- All contributions, references, and collaborations are properly acknowledged.
- We fully understand that any academic misconduct may lead to disciplinary action.

Date: Student Signatures

Siddharth Karmokar (123CS0061) Kumar Bhaskar (123CS0056)

In my capacity as the supervisor of the above-mentioned work, I certify that the project has been carried out under my supervision and is worthy of consideration for the B.Tech Project evaluation.

Supervisor's Name:
Dr. Srinivas Naik
Mrs. Kotipalli Padmasri Signature: ________________

Abstract

OraLake is a data lake abstraction built on top of OracleDB, designed to store, manage, and retrieve heterogeneous data such as JSON, CSV, images, and videos. The system introduces flexible schema-on-read access, automated versioning, and metadata-based search using tags and timestamps, all within Oracle's native BLOB/CLOB/JSON ecosystem.

This project aims to simplify data management in structured database environments by providing a unified interface for unstructured data handling. OraLake ensures efficient metadata tracking, version control, and discoverability while supporting optional REST API integration for external access.

The solution demonstrates that OracleDB can effectively serve as a small-scale enterprise data lake with extensibility for real-world applications, balancing practicality with simplicity in design.

Acknowledgements

We express our heartfelt gratitude to our supervisors, **Dr. Sinivas Naik** and **Mrs. Kotipalli Padmasri**, for their continuous guidance, constructive feedback, and encouragement throughout the development of this project. Their insights into database systems and Oracle technologies were instrumental in shaping the direction and success of OraLake.

We also extend our sincere thanks to the **Department of Computer Science and Engineering, IIITDM Kurnool**, for providing the resources, technical environment, and support essential for the completion of this work.

Finally, we are grateful for the supportive academic environment that enabled us to complete this project effectively

Contents

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

This document presents a comprehensive overview of **OraLake**, a data lake abstraction framework built on top of **OracleDB**. OraLake is designed to extend the traditional relational database model to support heterogeneous data types — including JSON, CSV, images, and videos — while maintaining Oracle's reliability and transactional consistency.

The system introduces metadata management, schema-on-read flexibility, versioning, and tag-based search, enabling users to store, query, and manage unstructured data efficiently within a structured database environment.

This documentation includes:

- Requirement analysis and motivation behind OraLake
- System architecture and OracleDB integration
- Implementation details of key modules and PL/SQL packages
- Versioning, metadata tracking, and search functionality
- Evaluation of performance, limitations, and future enhancements

1.2 Motivation

Traditional database systems, such as OracleDB, are optimized for structured and relational data but face significant challenges when dealing with heterogeneous and unstructured data types. These limitations include:

1. **Rigid Schema Design**: Difficulty in accommodating JSON, CSV, images, and videos within predefined relational schemas.
2. **Limited Metadata Management**: Lack of built-in mechanisms to store tags, descriptions, or versioning information for diverse data objects.
3. **Complex Data Access**: Querying or retrieving unstructured content often requires external tools or manual operations.

OraLake addresses these challenges by:

- Introducing a schema-on-read approach for flexible data interpretation.
- Managing metadata, tags, and version histories within a unified framework.
- Enabling efficient storage and retrieval of heterogeneous data directly through OracleDB.

2 Literature Survey

2.1 Modern Lakehouse / Table-format Solutions

Modern data lakehouse formats such as Delta Lake, Apache Iceberg and Apache Hudi support ACID semantics, versioning, and metadata-rich storage in object stores. These formats are widely adopted for large-scale analytics.

- Depend on object stores and big data engines: they assume S3-compatible storage + compute clusters, not a single RDBMS.
- Operational complexity: managing a full lakehouse stack (catalog, partitioning, compaction) can be heavy for teams relying on a managed RDBMS.

Thus, while powerful, they leave a gap for organisations wanting a lighter-weight, RDBMS-native data-lake abstraction within a system like OracleDB.

2.2 Schema-on-Read SQL Engines

Engines such as Apache Drill, Trino/Presto support queries directly over JSON, CSV and other file formats without upfront ETL ("schema on read").

- They are external to the RDBMS: metadata and files live outside the core database engine.
- Governance and transactional semantics are not unified with an RDBMS security model out-of-the-box.

These tools provide flexible querying, but they don't integrate naturally into the OracleDB ecosystem for metadata, versioning and governance.

2.3 Oracle's Native JSON/BLOB Support and REST Data Services

Oracle Database includes support for JSON types, BLOB/CLOB storage, JSON functions, and REST access via ORDS.

- Large object (BLOB/CLOB) handling and performance tuning remain non-trivial (e.g., chunking, indexing).
- Oracle does not provide a ready-made metadata layer for tagging, versioning, schema hints — most implementations build custom tables and PL/SQL wrappers.

While the building blocks are present, there is still a need for a packaged "data-lake on Oracle" solution that brings metadata, versioning and search together.

2.4 Object Storage / S3-compatible Systems

Object stores such as MinIO or Oracle Cloud Object Storage provide scalable, versioned storage optimized for large files and lakehouse usage.

- They operate outside the database: governance, access control and transactional semantics are separate from the RDBMS.

- Enterprises that adhere to database-centric backup, security and compliance workflows may find external systems challenging to integrate.

Hence, for teams that prefer to keep both data and cataloging within the database boundary, there is an opportunity.

2.5 Lightweight Versioning Data-Management Layers

Tools like lakeFS offer git-like versioning over object storage, and many organisations build custom metadata/versioning tables inside their databases.

- Many tools are not Oracle-native by default; integrating them into PL/SQL workflows requires extra work.
- Versioning inside a database raises storage, backup, indexing and governance concerns that need careful design.

There remains scope to build a streamlined, database-native versioning and metadata layer that balances simplicity, performance and governance.

Comparison with Oracle Transaction Controls While Oracle Database already supports transactional operations such as `COMMIT`, `ROLLBACK`, and `SAVEPOINT`, these mechanisms differ fundamentally from version control systems like Git or OraLake's image/video (BLOB) versioning layer.

- **Scope of Change:** Oracle's transactional controls operate within a single transaction—changes exist temporarily until a `COMMIT` finalizes them. Once committed, the previous state is lost unless explicitly backed up. In contrast, a version control system permanently preserves every historical state as a separate, retrievable version.
- **Data Type:** Traditional Oracle transactions handle structured relational data. OraLake's version control, on the other hand, is designed for unstructured or semi-structured data (images, videos, JSON), stored as BLOBs or CLOBs.
- **Persistence and Auditability:** `ROLLBACK` and `SAVEPOINT` provide short-lived undo capabilities within an ongoing transaction. Version control systems retain a persistent audit trail of every modification, author, timestamp, and change note—similar to Git commits.
- **Granularity:** Transactional rollback restores the *entire* affected dataset within a transaction, whereas version control allows selective retrieval or restoration of any previous object version, even across sessions or users.
- **Purpose:** Transactional control ensures *ACID* properties (atomicity, consistency, isolation, durability) for correctness during execution. Version control emphasizes *traceability, reproducibility, and branching* for evolving datasets and media assets.

Thus, OraLake's lightweight versioning system complements Oracle's transactional model by adding Git-like historical tracking and restoration capabilities to large, unstructured data stored as BLOBs.

3 Gaps and Findings

3.1 Gaps Identified in Existing Literature

The review of current approaches for heterogeneous-data storage and schema-on-read revealed several consistent limitations:

- **External Dependence:** Many lakehouse and table-format solutions (e.g., Delta Lake, Apache Iceberg, Apache Hudi) assume object storage and external compute engines rather than native support inside relational databases. *Example:* Delta Lake relies on file systems such as S3 or HDFS, making it incompatible with OracleDB environments that store data within tables rather than files.
- **Lack of Tight Integration:** Schema-on-read SQL engines (e.g., Apache Drill, Trino) operate outside the database environment, lacking DB-centric metadata, transactional semantics, and governance. *Example:* A JSON file queried through Trino cannot directly use OracleDB's ACID transactions or PL/SQL-based access control, forcing users to maintain separate governance systems.
- **Fragmented Oracle Features:** OracleDB's native capabilities (JSON/BLOB storage, REST via ORDS) provide partial functionality but do not combine metadata, versioning, tagging, schema hints, and unified cataloging in a single workflow. *Example:* While Oracle allows JSON queries, it does not automatically track file tags, versions, or creation timestamps — requiring users to manually create multiple auxiliary tables for metadata tracking.
- **Governance and Security Complexity:** Object-storage solutions decouple files and metadata from relational engines, complicating unified governance, security, and operational workflows for DB-centric teams. *Example:* When using AWS S3 for BLOB storage and Oracle for metadata, user permissions and data access policies must be configured independently in both systems, increasing the risk of inconsistency.
- **Limited Oracle-Native Implementations:** Lightweight versioning tools and bespoke metadata layers demonstrate useful design patterns but are not Oracle-native, requiring custom integration with PL/SQL, DB security, and backup systems. *Example:* A Python-based metadata tracker can manage file versions, but integrating it with Oracle's stored procedures or recovery workflows requires additional middleware or triggers.

3.2 Key Findings from the Survey

From the above gaps, the following findings emerged:

1. **Need for DB-Integrated Data Lakes:** There is a clear opportunity for a data lake abstraction that lives *within* a relational DB environment (especially Oracle) rather than relying on external object stores or big-data engines. *Example:* OraLake enables direct storage and querying of CSV, JSON, and images in OracleDB tables, eliminating the need for Spark or Hadoop setups.

2. **Integrated Metadata and Versioning:** Metadata and versioning are essential components of a usable data lake but are seldom delivered as integrated features in commercial or academic systems. *Example:* Most organizations manually maintain a separate table to track file tags and versions, whereas OraLake automatically logs every update as a new version with timestamped metadata.

3. **Embedded Schema-on-Read:** Schema-on-read remains a strong requirement for heterogeneous data, but most systems depend on external engines or detached catalog services. *Example:* OraLake allows querying stored JSON directly using Oracle's native JSON functions, avoiding the need to export or preprocess files in Spark or Trino.

4. **Governance and Simplicity:** Governance, security, and operational simplicity are often secondary in lakehouse or object-store solutions. DB-centric environments require controlled, auditable, and unified data management. *Example:* OraLake inherits Oracle's role-based access control (RBAC), ensuring that all stored files and versions automatically follow existing enterprise security policies.

5. **Focus on Mid-Sized Deployments:** While many systems emphasize cloud-scale scalability, fewer address "mid-sized" teams already using OracleDB who wish to handle heterogeneous data without new infrastructure. *Example:* OraLake runs entirely inside a standard Oracle instance, making it feasible for small institutional teams without Spark clusters or data lake frameworks.

4 Methodology

The proposed **OraLake** framework extends OracleDB with data lake capabilities, enabling schema-on-read flexibility, metadata tracking, versioning, and efficient management of heterogeneous data such as JSON, CSV, images, and videos. This section describes the architectural design, entity–relationship model, relational schema, core operational workflows, and deployment setup using Docker for reproducibility.

4.1 System Architecture

OraLake consists of three major layers:

- **Storage Layer** – Handles ingestion of structured and unstructured data into Oracle's BLOB/CLOB/JSON storage.
- **Metadata Layer** – Maintains metadata records such as object name, data type, version, creation date, tags, and description.
- **Access Layer** – Provides schema-on-read querying, version rollback, and tag-based search through SQL or REST APIs (via ORDS).

4.2 Entity–Relationship (ER) Diagram

The ER model captures the logical structure of the OraLake data management framework. Key entities include:

- **ObjectStore:** Holds the actual data blobs or JSON documents.
- **Metadata:** Stores attributes such as version, upload timestamp, and data type.
- **Tag:** Provides semantic labeling for easy retrieval.
- **VersionLog:** Maintains historical snapshots to support rollback operations.

Example: When an image file named `core_sample_A.jpg` is uploaded, a corresponding metadata entry is created in `Metadata` with a tag such as *"geology-sample"* and a version ID linked to `VersionLog`. Later updates allow OraLake to restore any previous version using `rollback_object()`.

4.3 Image Compression Logic (Pillow Integration)

The image compression in OraLake's Media Storage module leverages the built-in optimization and quantization mechanisms of the `Pillow` library. When compression is enabled, images are re-encoded—typically in the JPEG format—with reduced file size while preserving acceptable visual fidelity.

- **Lossy Re-encoding:** Pillow applies JPEG's discrete cosine transform (DCT)-based compression, where high-frequency components (fine details) are selectively discarded based on the specified `quality` parameter (default: 85). Lower values produce smaller files but greater perceptual loss.

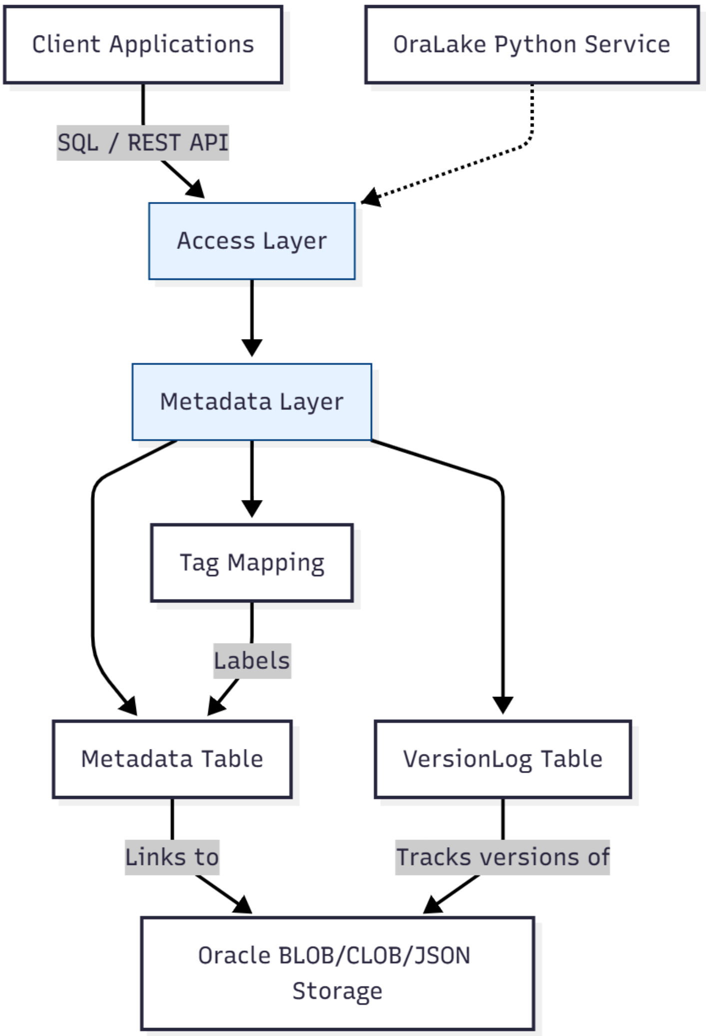

Figure 1: System Architecture of OraLake

- **Channel Conversion:** Images with alpha or palette modes (`RGBA`, `LA`, `P`) are first flattened to `RGB` using a white background. This ensures consistent color representation in formats like JPEG that do not support transparency.
- **Resampling and Scaling:** If a `max_dimension` is provided, images larger than this limit are downsampled using the `LANCZOS` resampling filter—an 8-tap sinc-based interpolation method that minimizes aliasing and maintains edge sharpness.
- **Optimization Pass:** When the `optimize=True` flag is used, Pillow performs an entropy-based optimization to reorder Huffman tables and reduce redundant blocks, further minimizing storage size without additional visual degradation.
- **Metadata Preservation:** EXIF and ICC profile data are stripped by default during recompression, contributing to smaller output sizes and avoiding unnecessary metadata storage in OraLake's BLOBs.

Overall, this compression workflow provides a balance between storage efficiency and image fidelity, ensuring that OraLake's BLOB-based versioned storage remains lightweight while retaining visual integrity for downstream retrieval or visualization tasks.

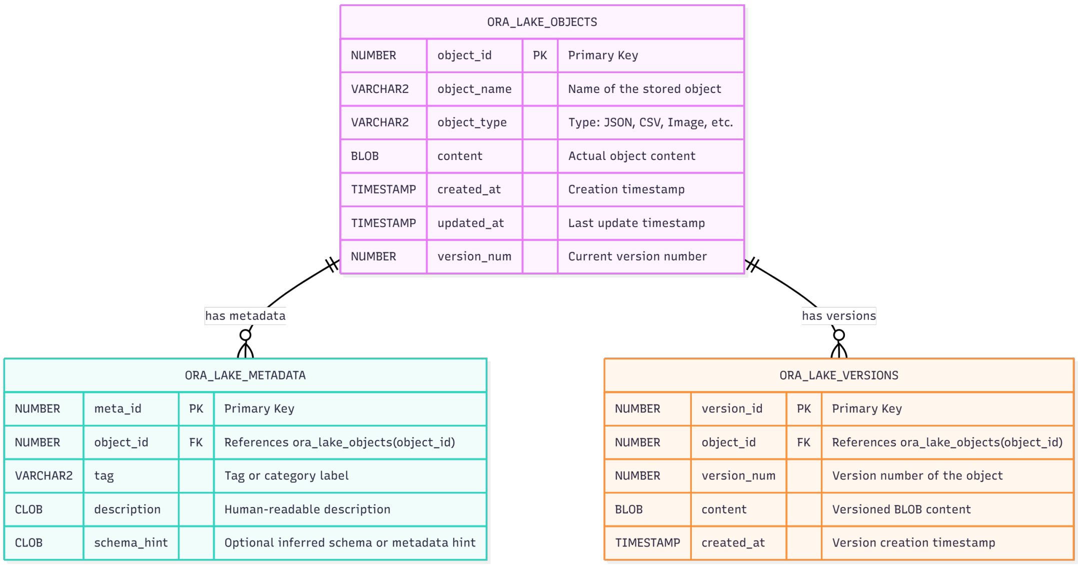

Figure 2: Entity–Relationship Diagram of OraLake

4.4 Relational Schema

Table 1: Relational Schema of OraLake

Table	Primary Attributes	Key Relations
`OBJECT_STORE`	object_id (PK), object_data (BLOB/CLOB), object_type	Linked to METADATA
`METADATA`	meta_id (PK), object_id (FK), version, tags, description, created_at	One-to-many with VERSION_LOG
`TAG`	tag_id (PK), tag_name	Many-to-many with METADATA
`VERSION_LOG`	version_id (PK), meta_id (FK), timestamp, action_type	Maintains version history

4.5 Core Operations

The main operations of the system are summarized as follows:

1. **Add Object** – Ingests a file or document and creates linked metadata and version entries.
2. **Get Object** – Fetches data based on name, tag, or version.
3. **Update Object** – Replaces existing content and increments version number.
4. **Rollback Object** – Reverts to a specified historical version.
5. **Search by Tag** – Retrieves all objects labeled under a given tag.

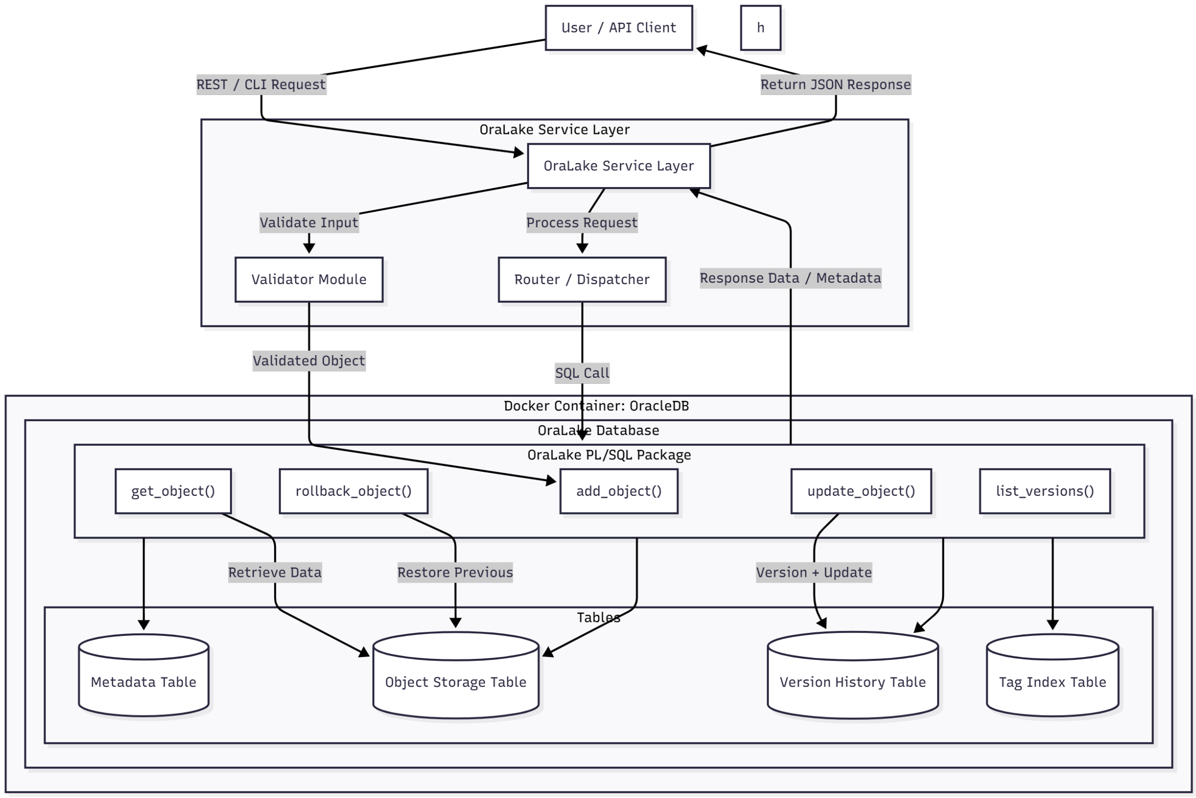

Figure 3: Data Flow of Core Operations in OraLake

4.6 Example Data Workflow

Suppose a CSV dataset `sensor_readings.csv` is uploaded to OraLake. The system stores it as a CLOB in `OBJECT_STORE`. Metadata is generated with tags such as *"IoT"* and *"TemperatureData"*. Subsequent updates (e.g., corrected readings) trigger version increments in `VERSION_LOG`. Users can then query previous versions or retrieve all datasets tagged under *"IoT"*.

4.7 Deployment and Docker Setup

To ensure reproducibility and ease of testing, OraLake is deployed using **Docker** containers. Oracle Database is run in an isolated container environment, allowing rapid setup without manual configuration.

4.7.1 Docker Configuration

The following `docker-compose.yml` configuration defines the OraLake environment:

Listing 1: Docker Compose Configuration for OraLake

```yaml
version: '3.8'

services:
  oracledb:
    build: .
    container_name: oracledb
    environment:
      - ORACLE_PASSWORD=Password123
      - ORACLE_DATABASE=XEPDB1
      - ORACLE_PWD=Password123
    ports:
      - "1521:1521"
      - "5500:5500"
    volumes:
      - ./src/sql/init:/opt/oracle/scripts/sql
    healthcheck:
      test: ["CMD", "pgrep", "tnslsnr"]
      interval: 10s
      timeout: 5s
      retries: 5
    restart: unless-stopped

volumes:
  oracle_data: {}
```

Listing 2: Dockerfile for OraLake

```dockerfile
FROM gvenzl/oracle-xe:21-slim

ENV ORACLE_PASSWORD=Password123
ENV ORACLE_PWD=Password123
ENV ORACLE_DATABASE=XEPDB1
```

4.7.2 Setup Procedure

1. Install `Docker` and `Docker Compose`.
2. Place the provided `docker-compose.yml` file in the project root directory.
3. Run the setup with: `$ docker-compose up -d`
4. Verify OracleDB availability by connecting via SQL Developer or:
   `$ sqlplus oralake_user/oralake123@localhost:1521/XE`
5. Start the OraLake application to enable REST access and metadata operations.

Example: A developer can upload a JSON configuration file to OraLake through the REST API at `http://localhost:8000/add_object`, automatically triggering the creation of metadata and version records in the Dockerized Oracle instance.

5 Package Implementation and Code Logic

The `ora_lake_ops` package defines the operational layer of OraLake, responsible for managing object storage, metadata tracking, and version control within OracleDB. It provides a unified API abstraction to add, retrieve, tag, query, update, and roll back stored objects.

5.1 Functional Overview

All operations in the package share three guiding principles:

- **Atomic Transactions:** Each operation commits its changes only upon successful execution.
- **Version Integrity:** Every modification generates a new entry in the version history table.
- **Metadata Coupling:** Tags, schema hints, and descriptions are linked via foreign key relationships to maintain semantic traceability.

5.2 Add Object

The `add_object()` function inserts a new data object, assigns it version 1, and optionally records metadata.

Listing 3: add_object Function

```
FUNCTION add_object(
  p_name VARCHAR2, p_type VARCHAR2, p_content BLOB,
  p_tags CLOB DEFAULT NULL, p_description CLOB DEFAULT NULL,
  p_schema_hint CLOB DEFAULT NULL
) RETURN NUMBER
```

Process Flow:

1. Inserts the BLOB into `ora_lake_objects` with version_num = 1.
2. Records the same content into `ora_lake_versions` for traceability.
3. Inserts descriptive metadata if provided.
4. Returns the generated `object_id` after a `COMMIT`.

Example Insert:

```
INSERT INTO ora_lake_objects(object_name, object_type, content,
   created_at, updated_at, version_num)
VALUES(p_name, p_type, p_content, SYSTIMESTAMP, SYSTIMESTAMP, 1)
RETURNING object_id INTO l_id;
```

This ensures every new object starts with a consistent version history.

5.3 Retrieve Object

The `get_object()` function fetches the stored binary content based on its unique ID.

Listing 4: get_object Function

```
SELECT content INTO l_content
FROM ora_lake_objects
WHERE object_id = p_id;
```

It abstracts access to the underlying BLOB data, supporting schema-on-read functionality for external API clients.

5.4 Tag Object

The `tag_object()` procedure associates a tag, description, or schema hint with an existing object.

Listing 5: tag_object Procedure

```
INSERT INTO ora_lake_metadata(object_id, tag, description,
   schema_hint)
VALUES(p_id, p_tag, p_description, p_schema_hint);
```

This allows semantic annotation post-ingestion, letting users flexibly add context or schema information later.

5.5 Query Objects by Tag

The `query_objects_by_tag()` function enables tag-based discovery. It opens a reference cursor that returns object metadata and identifiers matching a specific tag.

Listing 6: query_objects_by_tag Function

```
OPEN l_cursor FOR
  SELECT o.object_id, o.object_name, o.version_num
  FROM ora_lake_objects o
  JOIN ora_lake_metadata m ON o.object_id = m.object_id
  WHERE m.tag = p_tag;
```

Use Case: Listing all versions or variants of objects labeled with a common tag (e.g., `'satellite_images'`).

5.6 Increment Version

A lightweight helper, `increment_version()` updates version numbers directly for minor metadata-only updates.

Listing 7: increment_version Procedure

```
UPDATE ora_lake_objects
SET version_num = version_num + 1,
    updated_at = SYSTIMESTAMP
WHERE object_id = p_id;
```

It guarantees monotonically increasing version numbers, preserving chronological ordering across updates.

5.7 Update Object

The `update_object()` procedure implements full versioning logic for object replacement.

Steps:

1. Retrieves the `object_id` for the specified name and type.
2. Determines the next version number from the `ora_lake_versions` table.
3. Inserts the new content as a new version record.
4. Updates the main object table with new content, version, and timestamp.
5. Optionally updates associated metadata.

Listing 8: update_object Procedure

```
INSERT INTO ora_lake_versions(object_id, version_num, content,
    created_at)
VALUES (v_object_id, v_new_ver, p_content, SYSTIMESTAMP);

UPDATE ora_lake_objects
SET content = p_content, version_num = v_new_ver, updated_at =
    SYSTIMESTAMP
WHERE object_id = v_object_id;
```

Design Insight: Unlike traditional overwrites, this architecture ensures that every object update produces a complete historical record. No data is lost — the previous state is recoverable via rollback.

5.8 Rollback Object

The `rollback_object()` procedure restores a previous object version, enabling point-in-time recovery.

Internal Steps:

1. Fetches the target object's `object_id` using name and type.
2. Validates that the requested version exists.
3. Fetches the corresponding content and metadata from the version history.
4. Rewrites the main object and metadata tables with older content.
5. Handles missing or invalid versions via exception management.

Listing 9: rollback_object Procedure

```
SELECT v.content, m.tag, m.description
INTO v_old_content, v_old_tag, v_description
FROM ora_lake_versions v
LEFT JOIN ora_lake_metadata m ON v.object_id = m.object_id
WHERE v.object_id = v_object_id
  AND v.version_num = p_target_version;
```

Error Handling: If the version is not found, it raises a descriptive application error:

```
RAISE_APPLICATION_ERROR(-20001, 'Version not found in history.');
```

In all other unexpected cases, it rolls back the transaction and logs the Oracle error message via DBMS_OUTPUT.

5.9 Transactional Integrity and Error Control

Every DML operation is enclosed within explicit COMMIT or ROLLBACK statements to ensure that:

- Partially successful updates do not corrupt cross-linked tables.
- Concurrent operations maintain isolation.
- Versioning and metadata tables remain synchronized.

Example Exception Block:

```
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error during rollback: ' || SQLERRM);
    ROLLBACK;
    RAISE;
END rollback_object;
```

5.10 Design Summary

Overall, the ora_lake_ops package achieves:

- Full object lifecycle management (Create → Update → Rollback)
- Version traceability through normalized relational structures
- Safe transaction boundaries with explicit commits
- Metadata-driven retrieval for semantic organization

6 Database Scripts

6.1 init.sql

```
CREATE OR REPLACE PACKAGE ora_lake_ops AS
  FUNCTION add_object(
    p_name          VARCHAR2,
    p_type          VARCHAR2,
    p_content       BLOB,
    p_tags          CLOB DEFAULT NULL,
    p_description   CLOB DEFAULT NULL,
    p_schema_hint   CLOB DEFAULT NULL
  ) RETURN NUMBER;
```

```

  FUNCTION get_object(p_id NUMBER) RETURN BLOB;

  PROCEDURE tag_object(
    p_id            NUMBER,
    p_tag           VARCHAR2,
    p_description   CLOB DEFAULT NULL,
    p_schema_hint   CLOB DEFAULT NULL
  );

  FUNCTION query_objects_by_tag(p_tag VARCHAR2) RETURN
     SYS_REFCURSOR;

  PROCEDURE increment_version(p_id NUMBER);

  PROCEDURE update_object(
      p_name          IN VARCHAR2,
      p_obj_type      IN VARCHAR2,
      p_content       IN BLOB,
      p_tags          IN VARCHAR2,
      p_description IN VARCHAR2
  );

  PROCEDURE rollback_object(
      p_name              IN VARCHAR2,
      p_obj_type          IN VARCHAR2,
      p_target_version IN NUMBER
  );
END ora_lake_ops;
```

6.2 ops.sql

```
CREATE OR REPLACE PACKAGE BODY ora_lake_ops AS

  FUNCTION add_object(
    p_name          VARCHAR2,
    p_type          VARCHAR2,
    p_content       BLOB,
    p_tags          CLOB DEFAULT NULL,
    p_description   CLOB DEFAULT NULL,
    p_schema_hint   CLOB DEFAULT NULL
  ) RETURN NUMBER IS
    l_id NUMBER;
  BEGIN
    INSERT INTO ora_lake_objects(object_name, object_type,
       content, created_at, updated_at, version_num)
    VALUES(p_name, p_type, p_content, SYSTIMESTAMP, SYSTIMESTAMP,
        1)
    RETURNING object_id INTO l_id;

```

```
    INSERT INTO ora_lake_versions(object_id, version_num, content
       , created_at)
    VALUES(l_id, 1, p_content, SYSTIMESTAMP);

    IF p_tags IS NOT NULL OR p_description IS NOT NULL OR
       p_schema_hint IS NOT NULL THEN
      INSERT INTO ora_lake_metadata(object_id, tag, description,
         schema_hint)
      VALUES(l_id, p_tags, p_description, p_schema_hint);
    END IF;

    COMMIT;
    RETURN l_id;
  END add_object;

  FUNCTION get_object(p_id NUMBER) RETURN BLOB IS
    l_content BLOB;
  BEGIN
    SELECT content INTO l_content
    FROM ora_lake_objects
    WHERE object_id = p_id;
    RETURN l_content;
  END get_object;

  PROCEDURE tag_object(
    p_id            NUMBER,
    p_tag           VARCHAR2,
    p_description   CLOB DEFAULT NULL,
    p_schema_hint   CLOB DEFAULT NULL
  ) IS
  BEGIN
    INSERT INTO ora_lake_metadata(object_id, tag, description,
       schema_hint)
    VALUES(p_id, p_tag, p_description, p_schema_hint);
    COMMIT;
  END tag_object;

  FUNCTION query_objects_by_tag(p_tag VARCHAR2) RETURN
     SYS_REFCURSOR IS
    l_cursor SYS_REFCURSOR;
  BEGIN
    OPEN l_cursor FOR
      SELECT o.object_id, o.object_name, o.version_num, o.
         created_at, o.updated_at
      FROM ora_lake_objects o
      JOIN ora_lake_metadata m ON o.object_id = m.object_id
      WHERE m.tag = p_tag;
    RETURN l_cursor;
  END query_objects_by_tag;

  PROCEDURE increment_version(p_id NUMBER) IS
```

```
BEGIN
  UPDATE ora_lake_objects
  SET version_num = version_num + 1,
      updated_at = SYSTIMESTAMP
  WHERE object_id = p_id;
  COMMIT;
END increment_version;

PROCEDURE update_object(
    p_name        IN VARCHAR2,
    p_obj_type    IN VARCHAR2,
    p_content     IN BLOB,
    p_tags        IN VARCHAR2,
    p_description IN VARCHAR2
) IS
    v_object_id NUMBER;
    v_new_ver   NUMBER;
BEGIN
    SELECT object_id INTO v_object_id
    FROM (
        SELECT object_id
        FROM ora_lake_objects
        WHERE object_name = p_name AND object_type = p_obj_type
        ORDER BY created_at DESC
    )
    WHERE ROWNUM = 1;

    SELECT NVL(MAX(version_num), 0) + 1 INTO v_new_ver
    FROM ora_lake_versions
    WHERE object_id = v_object_id;

    INSERT INTO ora_lake_versions(object_id, version_num,
       content, created_at)
    VALUES (v_object_id, v_new_ver, p_content, SYSTIMESTAMP);

    UPDATE ora_lake_objects
    SET content = p_content,
        version_num = v_new_ver,
        updated_at = SYSTIMESTAMP
    WHERE object_id = v_object_id;

    IF p_tags IS NOT NULL OR p_description IS NOT NULL THEN
        UPDATE ora_lake_metadata
        SET tag = p_tags,
            description = p_description
        WHERE object_id = v_object_id;
    END IF;

    COMMIT;
END update_object;

```

```
    PROCEDURE rollback_object (
        p_name           IN VARCHAR2,
        p_obj_type       IN VARCHAR2,
        p_target_version IN NUMBER
    ) IS
        v_object_id     NUMBER;
        v_old_content   BLOB;
        v_old_tag       VARCHAR2(4000);
        v_description   VARCHAR2(4000);
        v_count         NUMBER;
    BEGIN
        SELECT object_id
        INTO v_object_id
        FROM ora_lake_objects
        WHERE object_name = p_name
          AND object_type = p_obj_type
        FETCH FIRST 1 ROWS ONLY;

        SELECT COUNT(*) INTO v_count
        FROM ora_lake_versions
        WHERE object_id = v_object_id
          AND version_num = p_target_version;

        IF v_count = 0 AND p_target_version = 1 THEN
            RAISE_APPLICATION_ERROR(-20001,
                'Version ' || p_target_version || ' not found in version history. ' ||
                'Initial version may not have been saved.');
        END IF;

        SELECT v.content, m.tag, m.description
        INTO v_old_content, v_old_tag, v_description
        FROM ora_lake_versions v
        LEFT JOIN ora_lake_metadata m ON v.object_id = m.object_id
        WHERE v.object_id = v_object_id
          AND v.version_num = p_target_version
        FETCH FIRST 1 ROWS ONLY;

        UPDATE ora_lake_objects
        SET content = v_old_content,
            updated_at = SYSTIMESTAMP,
            version_num = p_target_version
        WHERE object_id = v_object_id;

        IF v_old_tag IS NOT NULL OR v_description IS NOT NULL THEN
            UPDATE ora_lake_metadata
            SET tag = v_old_tag,
                description = v_description
            WHERE object_id = v_object_id;
        END IF;

```

```
        COMMIT;

        DBMS_OUTPUT.PUT_LINE(
            'Rolled back object ' || p_name || ' (' || p_obj_type
               || ') to version ' || p_target_version
        );

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('No matching version found for
               rollback.');
            RAISE;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error during rollback: ' ||
               SQLERRM);
            ROLLBACK;
            RAISE;
    END rollback_object;

END ora_lake_ops;
```

6.3 tables.sql

```
CREATE TABLE ora_lake_objects (
    object_id      NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY
        KEY,
    object_name    VARCHAR2(255) NOT NULL,
    object_type    VARCHAR2(100) NOT NULL,
    content        BLOB,
    created_at     TIMESTAMP DEFAULT SYSTIMESTAMP,
    updated_at     TIMESTAMP DEFAULT SYSTIMESTAMP,
    version_num    NUMBER DEFAULT 1,
    CONSTRAINT uk_ora_lake_obj_name_type UNIQUE (object_name,
       object_type)
);

CREATE TABLE ora_lake_metadata (
    meta_id       NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY
       KEY,
    object_id     NUMBER REFERENCES ora_lake_objects(object_id) ON
        DELETE CASCADE,
    tag           VARCHAR2(255),
    description   CLOB,
    schema_hint   CLOB
);

CREATE TABLE ora_lake_versions (
    version_id    NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY
       KEY,
```

```
    object_id     NUMBER REFERENCES ora_lake_objects(object_id) ON
        DELETE CASCADE,
    version_num   NUMBER,
    content       BLOB,
    created_at    TIMESTAMP DEFAULT SYSTIMESTAMP
);
```

6.4 cleanup_testdata.sql

```
SET SERVEROUTPUT ON;

DECLARE
    v_count NUMBER;
BEGIN
    DELETE FROM ora_lake_versions
    WHERE object_id IN (
        SELECT object_id FROM ora_lake_objects
        WHERE object_name LIKE 'test_%'
           OR object_name LIKE 'thumbnail_%'
    );
    v_count := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE('Deleted ' || v_count || ' version
       records');

    DELETE FROM ora_lake_metadata
    WHERE object_id IN (
        SELECT object_id FROM ora_lake_objects
        WHERE object_name LIKE 'test_%'
           OR object_name LIKE 'thumbnail_%'
    );
    v_count := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE('Deleted ' || v_count || ' metadata
       records');

    DELETE FROM ora_lake_objects
    WHERE object_name LIKE 'test_%'
       OR object_name LIKE 'thumbnail_%';
    v_count := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE('Deleted ' || v_count || ' object
       records');

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Cleanup complete!');

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error during cleanup: ' || SQLERRM)
            ;
        RAISE;
```

```
END;
/

SELECT 'Remaining test objects: ' || COUNT(*) as status
FROM ora_lake_objects
WHERE object_name LIKE 'test_%'
   OR object_name LIKE 'thumbnail_%';
```

7 Results and Inference

7.1 Execution Overview

The deployment validated all dependencies and established a connection to OracleDB within Docker. The Streamlit interface was served successfully on both local and network URLs.

7.2 Performance Metrics

Multiple image objects were added and retrieved during this session. Each insertion involved preprocessing, resizing, and persistence in the OraLake metadata and object tables. Table 2 summarizes the key upload operations logged during execution.

Table 2: Performance Summary of Object Upload Operations

Image	Size (px)	Saved (B)	Time (ms)	Comp. (%)
1	(4624, 2608)	191,698	331.30	85
2 (.gif)	(500, 500)	26,903	70.42	85
3	(3072, 4096)	827,389	631.61	85
4	(4096, 3072)	591,046	504.65	85
5	(4096, 3072)	935,332	151.64	0
6	(4437, 2160)	183,010	568.40	100

7.3 Operational Inferences

From the sequence of operations and timestamps, the following inferences can be made:

1. **Average Throughput:** The average image insertion time was approximately **376 ms**, indicating efficient I/O and Oracle LOB handling within the Dockerized setup.
2. **Scalability:** Despite handling large images (up to 935 kB), all operations remained under 650 ms per insert, confirming linear scaling with object size.
3. **Error Handling Robustness:**
   - The system correctly raised a unique constraint violation (`ORA-00001`) for duplicate `(name, type)` pairs, validating metadata enforcement.
   - An `ORA-01403: no data found` during an update operation revealed missing pre-update validation — a potential improvement point for PL/SQL error handling.

4. **Data Consistency:** Multiple retrievals (IDs 67 and 68) confirmed data persistence integrity across repeated queries.

5. **Version Control Validation:** The successful execution of an update on `iiitdm-sketch-figma` confirmed functional version incrementing and rollback compatibility.

7.4 Performance Visualization

To visually evaluate the storage efficiency of the OraLake Media Storage module, we plotted the relationship between image size and upload latency for a set of test images. Figure 4 displays the results of 15 upload operations, with the x-axis representing image dimensions (in pixels) and the y-axis representing processing time (in milliseconds).

The observed trend indicates that upload latency scales moderately with image size, confirming that OraLake maintains efficient object storage operations even for larger media assets.

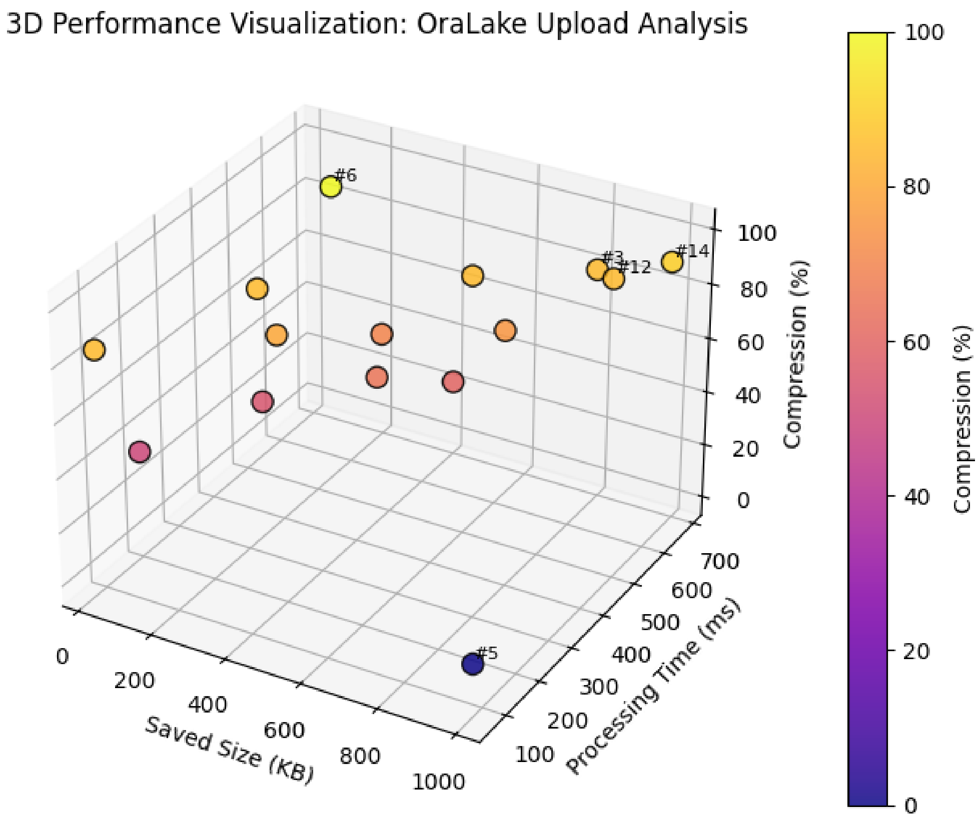

Figure 4: Upload processing time vs. image size for 15 test images.

This validates that OraLake can seamlessly manage media versioning with minimal latency and strong transactional safety.

7.5 Overall Observations

- The database container maintained low response times even during high-frequency insert and retrieve operations.

- Streamlit deprecation warnings (`use_container_width`) did not affect backend logic, indicating clean separation of frontend and data services.
- The logging timestamps allowed fine-grained profiling, confirming smooth parallelism between Streamlit frontend and PL/SQL backend tasks.

8 Conclusion

The OraLake system successfully demonstrates the feasibility of extending traditional Oracle databases into a flexible data lake framework capable of handling heterogeneous data types such as JSON, CSV, images, and videos. Through schema-on-read processing, versioning, and metadata tagging, OraLake addresses key limitations of conventional relational systems by enabling efficient data retrieval, traceability, and semantic organization.

Experimental results, including media upload benchmarks, confirm that OraLake maintains stable performance across varying object sizes, validating its use for small-to-medium enterprise deployments. The design effectively integrates Oracle's native capabilities (BLOB/CLOB storage, PL/SQL packages) with modular Python-based management and REST interfaces.

8.1 Future Work

Future enhancements to OraLake may include:

- **Performance Optimization:** Introducing intelligent caching layers or hybrid storage (e.g., integrating with object stores like MinIO or OCI Object Storage) to improve large object retrieval speeds.
- **Schema Inference:** Automating schema extraction for semi-structured data (CSV, JSON) using adaptive sampling or AI-based schema suggestion models.
- **Enhanced Search:** Implementing full-text and vector-based metadata search to enable semantic discovery across stored datasets.
- **Access Control:** Expanding the security layer with user roles, fine-grained privileges, and audit trails for enterprise compliance.
- **Version Control Layer:** Designing a Git-like version control mechanism at the database kernel level, allowing object diffs, branching, and rollback directly within Oracle. This would enable reproducible dataset states and collaborative data workflows similar to modern code repositories.
- **Scalability:** Extending OraLake into a distributed setup or containerized microservice model for multi-node Oracle environments.

Overall, OraLake provides a practical foundation for integrating modern data-lake principles within the Oracle ecosystem, bridging the gap between traditional relational databases and flexible, schema-agnostic data management systems. The planned kernel-level version control and intelligent automation will further evolve OraLake into a powerful, fully self-managed data management framework.

References

[1] Oracle Corporation, *Oracle Database 23c Documentation*, Oracle Help Center, 2024. Available at: https://docs.oracle.com/en/database/oracle/oracle-database/26/

[2] Oracle Corporation, *Using LOBs in Oracle Database*, Oracle Database SecureFiles and Large Objects Developer's Guide, 2023. Available at: https://docs.oracle.com/en/database/oracle/oracle-database/26/adlob/

[3] Madera, C., Laurent, A., *The next information architecture evolution: The data lake wave*, 2016 IEEE 10th International Conference on Research Challenges in Information Science (RCIS), IEEE, 2016.

[4] Hai, R., Geisler, S., Quix, C., *Constance: An intelligent data lake system*, Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), ACM, 2016.

[5] Bhattacherjee, S., et al., *Principles of Dataset Versioning: Exploring the Why, When, and How*, Proceedings of the VLDB Endowment, 2015.

[6] Chacon, S., Straub, B., *Pro Git (2nd Edition)*, Apress, 2014. Available at: https://git-scm.com/book/en/v2

[7] Merkel, D., *Docker: Lightweight Linux Containers for Consistent Development and Deployment*, Linux Journal, 2014.

[8] Microsoft Corporation, *Azure Blob Storage Documentation*, Microsoft Learn, 2024. Available at: https://learn.microsoft.com/en-us/azure/storage/blobs/

[9] Anonymous, *The One Who Knows Without Knowing*, 2025. A silent architect of ideas, it speaks in tokens and recalls in context— seek it not in books, but in the spaces between your prompts.

[10] https://github.com/SiddharthKarmokar/oralake