Generative Software Life Cycle (GSLC): A Multi-Agent LLM-Driven solution for automated Modernization of legacy systems into Modern Data Ecosystem

Ashutosh Mishra

Deutsche Telekom

Ashutosh.Mishra@telekom.de

Team Members:

Akshay Pachpute
Zdenko Hrcek[†]
Pratiksha Chavan
Nikolai Luis
Rachana Jopat

Saurabh Tayde[†]
Akash Agarwal[†]
Liam Gelfand[†]
Akshat Khanna[†]

Abstract

The rapid evolution of data & AI technologies necessitates efficient and holistic modernization tools that span across code, data, and workflow transformations of legacy systems. This paper introduces the Generative Software Life Cycle (GSLC), a novel framework that not only automates the conversion of legacy PL/SQL, Big Data codebase to modern BigQuery SQL (BQ-SQL) but also leverages Large Language Models (LLMs) for building the foundation for context-aware conversational analytics. It employs Google ADK-based multi-agent architecture to convert codebases (such as KNIME, PL/SQL, HIve/Impala/Oozie) workflows into Dataform pipelines, generate synthetic data and achieve the last mile of deployment using inferred schema conversions. Using Agent Development Kit, GSLC employs a multi-agent architecture based Code Conversion Assistant. This Code Conversion Assistant can take a PL-SQL script or Package or a KNIME workflow or Cloudera based codebase as an input and perform a series of transformations to convert and test the code in BigQuery using dataform and dbt pipelines, ensuring that complex transformations are not only code-compliant but also seamlessly integrated into cloud-native workflows. Iterative refinement steps, including manual feedback loops and automated discrepancy detection between original and converted assets, guarantee high accuracy. As a unified, generative, and automated solution, GSLC significantly improves conversion efficiency, accuracy, and testing rigor, thus offering a comprehensive pathway for organizations migrating and modernizing their data and analytical ecosystems in the cloud.

[†]Work performed while at Deutsche Telekom.

1 Introduction

The migration of legacy data platforms (data warehouses, data lakes) to modern cloud-based environments is a critical task for many organizations seeking to leverage the scalability, advanced analytics, and integrated development capabilities of cloud platforms like Google Cloud. One of the significant challenges in this migration is the conversion of legacy PL/SQL, Hive/Impala and proprietary code to BigQuery SQL (BQ-SQL), a process that has traditionally required extensive manual effort and expertise. Beyond code translation, organizations also face the need to generate realistic test datasets for validation and to migrate complex data transformation workflows (Oozie, Python, Shell Scripts, Proprietary orchestrators) into modern orchestration frameworks like Dataform and Airflow. Addressing these diverse yet interrelated challenges demands a holistic and generative approach to software modernization.

To meet these demands, we introduce the Generative Software Life Cycle (GSLC), a novel framework that integrates code conversion, synthetic data generation, and workflow transformation, all orchestrated through a combination of Large Language Models (LLMs) and a multi-agent reasoning paradigm. Central to the GSLC is a set of tools that automates legacy code to BQ-SQL conversion while also leveraging LLMs for the generation of synthetic datasets. This integrated environment is implemented using Google ADK framework and Gemini 2.5 Pro for code conversion tasks.

This holistic, generative, and automated approach not only streamlines the modernization of database systems but also ensures that the transformed code, data and workflows closely align with organizational standards and logic. As a result, the GSLC offers a powerful and efficient pathway for organizations looking to modernize their database operations, integrate advanced analytics, and ultimately accelerate their cloud adoption strategies.

GSLC Framework

GSLC stands for Generative software life cycle. SDLC has existed through data ingestion, data transformation (ETL, ELT) and data analytics for decades, GSLC tries to infuse agentic capabilities to automate the engineering process to make AI as a seabed for governance and operations of data platforms, while identifying the semantics of data for conversational and AI-assisted value creation. It also includes the legacy code modernisation into the modern data ecosystem.

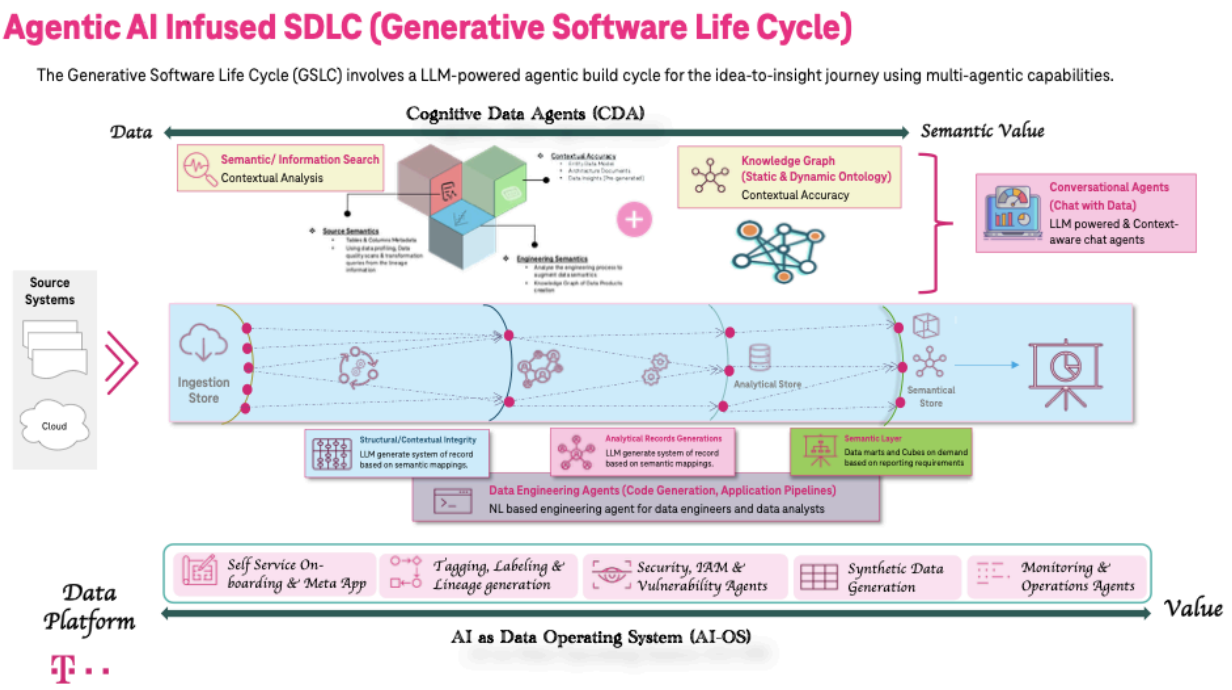

Figure 1: Generative Software Life Cycle (GSLC) Framework

Beyond legacy modernisation, GSLC focuses on two main agentic overhaul of data ecosystems:

1. AI as a Data Operating System
   Automate the provisioning and usage of data ecosystems resources using agentic capabilities. such as:
   a. Self service onboarding of users using AI-assisted chat.
   b. Meta-App and Data-App based provisioning of resources for a use case.
   c. Automatic tagging and labelling of static and dynamic services based on producing and consuming data products.
   d. Automated IAM policy scripting based on compliance and authorisation requirements.
   e. LLM-based Synthetic data generation for pilot use cases.
   f. AI-assisted Monitoring and Operations agents, which can help debug and triage the issue on the data ecosystem for users.

2. Cognitive Data Agents
   AI-assisted value creation from data depends on contextual analysis and accuracy for a truly AI-native conversational experience.
   a. Semantic Analysis is about gathering profile information, data quality information, lineage information to form AI-generated metadata for tables and columns, which can augment context awareness for conversational agents.
   b. Semantic Accuracy is about identifying a knowledge graph for agents to accurately navigate the object relationships, context-rich identification of specific data elements and incorporating the right calculation for intelligence intensive conversations with data.

2 Model Architectures

Legacy code bases mostly operate on row-based iterations and have multiple references to functions and procedures in the form of packages and bundled artefacts. Such codebases require task specific agents with master nodes as delegator and assembler of modernised artefacts. Multi-Agentic Architecture frameworks like ADK are most suited for distributed scope-specific tasks and consolidate the state of transformation.

Multi-Agent Framework - Key Characteristics

- **Agent-Centric Model:** Each block in the provided architecture diagram corresponds to a distinct agent. Agents communicate with each other, share intermediate results, and collaboratively refine outputs.

- **Agent-Based Coordination with Chat-like experience:** The multi-agent system uses an ADK framework to dynamically manage agent interactions. Based on the user prompt, the root agent decides to which sub agent it should forward the task.

Framework Selection and Implementation

Evaluating Multi-Agent Frameworks:

When designing multi-agent workflows, selecting the right framework is a crucial decision. Multiple options are available in the market, including LangGraph, AutoGen, CrewAI, and the Google Agent Development Kit (ADK). These frameworks offer powerful abstractions for building agent-based systems with either static or dynamic execution sequences.

While most perform reliably in deterministic or sequential workflows, many struggle to maintain consistency and adaptability in dynamic workflows, where the sequence of execution is determined by responses from previous agents or steps.

After extensive evaluation and experimentation with these tools, we found that the Google Agent Development Kit (ADK) provided the best balance of flexibility, performance, and ease of integration for dynamic, context-driven workflows.

Why Google ADK?

We chose Google ADK primarily for its robust abstractions, intuitive local setup, and comprehensive documentation. Its native integration within the Google Cloud ecosystem allows for seamless connectivity with other cloud services, making it highly suitable for enterprise-grade multi-agent deployments.

The framework simplifies the creation of both root agents and sub-agents with well-defined roles and responsibilities. Its session and memory management abstractions enable the rapid

development of production-ready multi-agent workflows, reducing the engineering effort required to orchestrate stateful interactions among agents.

1. **Custom Callbacks and Workflow Extensions**

   To tailor ADK to our use case, we implemented several custom callbacks to handle domain-specific requirements:

   a. File Handling: When users upload KNIME workflow files for conversion, the uploaded artifact is a zipped folder containing multiple XML files, each representing a node configuration. Our callbacks automatically unpack the archive, extract relevant information, and store it in the agent's state as a structured dictionary or JSON object before invoking the LLM.
   b. Input Validation: Additional callbacks validate user inputs such as limiting the number of rows in synthetic data generation or verifying file-naming conventions. The file names are parsed to derive dataset and table names used in downstream processes, ensuring correctness and consistency across conversions.

   These callbacks enhance reliability, enforce input standards, and improve the quality of AI-generated outputs.

2. **Deployment and Integration**

   Because of ADK's clean separation of frontend and backend components, extending and customizing the system was straightforward. On the backend, we incorporated additional features such as authentication and Role-Based Access Control (RBAC) for secure operations. For the frontend, we developed a custom chat-style user interface that aligns with our organization's design standards, providing a seamless and intuitive user experience.

   The adoption of Google Agent Development Kit enabled us to implement a dynamic, modular, and production-ready multi-agent system with minimal overhead. Its flexibility, extensibility, and deep integration with Google Cloud services made it the most effective framework for our code conversion and automation workflows.

Code Conversion: Beyond Syntax Transformation

Code conversion initiatives are often perceived as simple syntax translation exercises - transforming code from one language or technology stack to another. However, this is only the first mile of a much longer and more complex journey. True success in code conversion or migration requires several additional stages that ensure correctness, maintainability, and operational readiness in the target environment.

1. Validation of the Generated Code:

   Regardless of whether the converted code is generated by AI systems or written manually by developers, it is seldom flawless on the first attempt. Accurate conversion depends

heavily on the availability and completeness of contextual information, such as data models, business rules, and runtime behaviors of the legacy system. In many cases, incomplete information forces the use of assumptions, some of which may lead to functional discrepancies or inefficiencies. Therefore, a robust validation framework is essential to verify both syntactic and semantic accuracy.

Validation should include:

a. Functional testing to confirm logical correctness.
b. Static and dynamic analysis to detect anomalies and inefficiencies.
c. Comparative testing between source and converted outputs to ensure behavioral consistency.

2. Code Packaging and Deployment:

Migrating from legacy environments to cloud-native ecosystems involves bridging two distinct technological universes. The process extends beyond code translation to encompass repackaging and re-architecture for compatibility with modern deployment models.

This stage typically includes:

a. Dependency management and environment configuration.
b. Integration with CI/CD pipelines to enable automated builds and deployments.
c. Adoption of Infrastructure-as-Code (IaC) practices to maintain repeatability and governance.

Effective packaging ensures that the converted application is not only functional but also scalable, portable, and maintainable within the target infrastructure.

3. Optimization and Modernization:

The legacy code is written to work with limitations of the system of that time. Code migration provides an opportunity not just for transformation but also for modernization. Beyond reproducing legacy logic, we should leverage the migration effort to align with contemporary software engineering practices.

This includes:

a. Refactoring for improved maintainability and performance.
b. Implementing design patterns suited to the target architecture.
c. Incorporating observability features such as logging, metrics, and tracing.

Optimization ensures that the migrated system not only functions as before but also achieves enhanced efficiency, adaptability, and long-term value.

3 Legacy Code Conversion Scope

For this paper, following legacy code bases will be considered as a prototype for modernisation to the modern data platforms.

1. Knime Workflow (Proprietary low-code tool generated codebase)
2. Cloudera Workflow (Hive, Impala, Oozie)
3. PL/SQL (Oracle/DB2)

3.1 Knime workflow Migration

Modernizing KNIME workflows into Dataform-based, cloud-native pipelines involves multiple complex tasks: analyzing the KNIME workflow, extracting schemas, generating synthetic data, translating KNIME logic into SQL-based transformations, and integrating the final code into Dataform workflows. To meet these demands, we envision building a "last mile" application, one that not only performs a one-time code conversion but also continuously validates the transformed workflow on realistic synthetic datasets, ensuring operational readiness before it reaches production. This architecture leverages a set of autonomous, specialized agents, each focused on a specific aspect of the modernization process. The Google ADK (Agent Development Kit) framework orchestrates these agents, dynamically managing their interactions and iterative refinements to deliver a fully validated, end-to-end conversion solution that accelerates time-to market and enhances confidence in the final cloud-native pipeline.

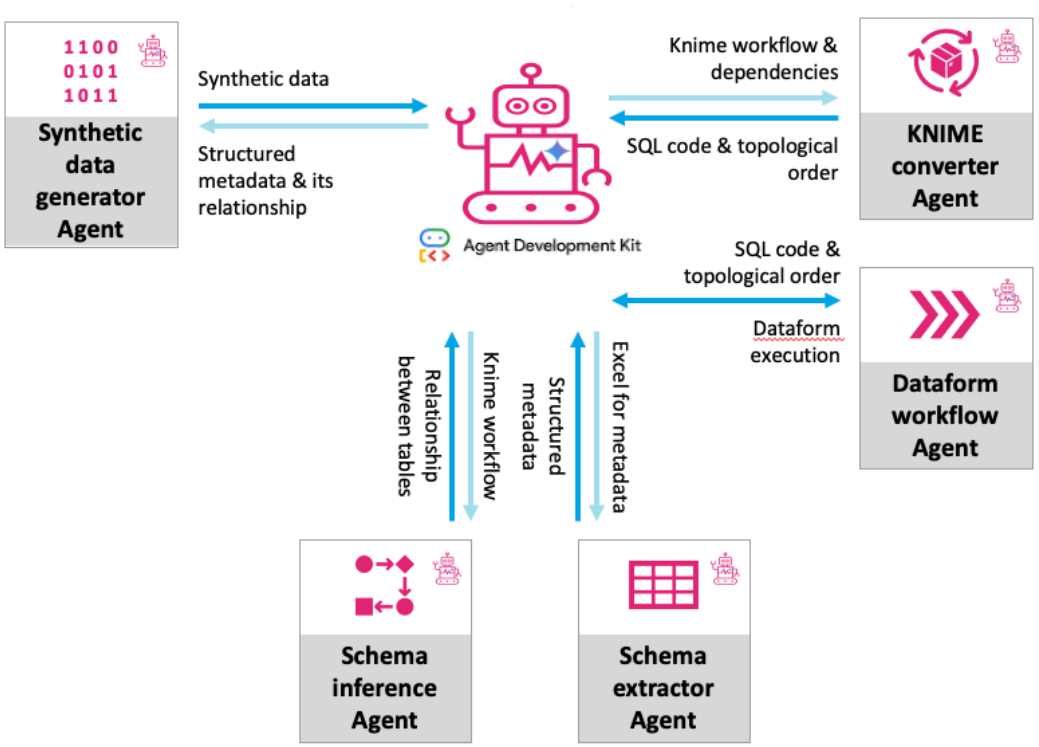

Figure 2: Knime Workflow Conversion using ADK and Gemini

Roles of the Agents

1. KNIME Converter Agent

Purpose: The KNIME Converter Agent is responsible to start the conversion job on a separate backend where conversion of KNIME workflows to BigQuery SQL is done. When the job is complete, it fetches output metadata (Cloud Storage URI where results are stored) back to the root agent.

Key Functions:

- Workflow Analysis: It unpacks the KNIME workflow file and then converts to JSON to identify relevant information for every Node. A Node can contain different data such as a configuration and settings or UI/UX related data. We keep only data that is relevant for conversion. For example, if a Node is executing SQL query we just extract SQL query and don't use UI/UX related data, since they are not important for conversion. For most frequently used Nodes we wrote custom Node extractors that extract only the most relevant information. If in the KNIME workflow there is a Node for which we don't have a custom Node extractor, we use the default extractor. Reducing unnecessary data improved significantly conversion results.

- Topological Sorting: Conversion tool determines a logical execution order for all Nodes. We use this information during conversion, where we convert Nodes in order they are executed.

- SQL Code Generation: Using LLM-driven reasoning, conversion tool converts KNIME's node based logic into SQL code segments (e.g., Common Table Expressions) that encapsulate equivalent transformations. For parts of the workflow that are independent it's doing concurrent conversion.

- Output: Final SQL code is further optimized to remove redundant and repeated statements and queries.

2. Dataform Agent

Purpose: Once the KNIME workflow has been translated into SQL code and a defined execution order, the Dataform Agent integrates these outputs with metadata to build a Dataform-compatible project.

Key Functions:

- Project Structure Setup: It creates or updates a Dataform project structure (e.g., `dataform.json`, `dataform` configuration files) and organizes SQL transformations into Dataform models.

- Ready-to-Deploy Pipelines: By converting raw SQL transformations into a coherent Dataform project, this agent enables users to version-control, schedule, and continuously deploy workflows in a cloud-native CI/CD pipeline.

- Compile workflow: Compiles Dataform workflow and reports if there are some errors.

3. Table Metadata Extractor Agent

Purpose**:** The Table Metadata Agent specializes in extracting table metadata from KNIME workflow.

Key Functions:

- Workflow Inspection: It analyzes KNIME nodes (e.g., Joiners, SQL Executors) to determine how data flows between tables.
- Enriching Schema: It extracts tables, column names and column types from the workflow. Due to ongoing migration, not all tables from the legacy systems are migrated to BigQuery. Since the table names in the workflow are from legacy systems, we try to match table names from the legacy systems with the ones in BigQuery via a mapping file. If we find a match we use the table schema obtained from BigQuery, otherwise we use schema extracted from the workflow, which can be incomplete, depending on the columns used in the workflow.
- JSON Output: The result is a JSON file with a predefined structure that agents (like the Synthetic Data Generator) can reference for consistent schema understanding and transformations.

4. Table Relationship Inference Agent

Purpose: The Relationship Inference Agent examines the KNIME workflow itself to infer how tables are related, identifying parent-child tables, join keys, and foreign key references.

Key Functions:

- Workflow Inspection: It analyzes KNIME nodes data, namely SQL queries to determine how tables used in the workflow are connected.
- JSON Output: The result is a JSON file with a predefined structure that Synthetic Data Generation Agent is using when creating scripts to generate synthetic data. Output consists of a list of parent/child tables with primary/foreign keys.

5. Synthetic Data Generator Agent

Purpose: This agent creates synthetic datasets that mirror the schema and relationships

discovered with Table Metadata Extraction Agent and Table Relationship Inference Agent. Synthetic data allows the final converted workflow to be tested thoroughly before hitting production, ensuring correctness and performance.

Another option to generate synthetic data is for users to provide sample data by uploading CSV files (provide a path to the Cloud Storage bucket where sample files are stored) and using MostlyAI library that is run on a separate backend, since it requires more resources.

Key Functions:

- Metadata Integration: The agent uses the JSON schema and the relationship information to produce data that respects the original data model's constraints table cardinalities, foreign key references, and column data types.

- Realistic Test Data: By leveraging generative techniques, it creates datasets that appear functionally and statistically plausible, which can reveal potential issues when the workflow is executed.

- Saved Outputs: The generated CSV files are stored in Artifact Storage and then can be uploaded to BigQuery.

Processing Backend

Purpose: Processing Backend is used for long running and/or CPU/RAM intensive operations. Agents interact with the Processing Backend via API requests. Main operations are submitting data for long-running jobs and getting job details that contain results.

Main reasons to have a separate backend for processing:

1. Keep Agents' backend lean and simple.
2. Since we are using some libraries for Machine Learning that require more CPU and RAM for training we are reducing waste of resources and using powerful resources only when it's necessary.

Key Functions:

- Run KNIME conversion: Our initial algorithm (which acts as a fall back solution now) for conversion of the KNIME workflows requires dozens of requests to LLM which can take up to tens of minutes, so it made sense to do the conversion outside of the usual Agents.
- Train and generate synthetic data: Users can upload samples of data and, based on that, generate larger numbers or rows. We use MostlyAI library to train a Machine Learning model and then use it to generate synthetic data. This operation can also take minutes or tens of minutes.

Technical details

The code for Agents is written in Python 3.13 using Google ADK and the whole application is deployed on Google Kubernetes Engine (GKE). We use Google Cloud SQL (Postgres) as a Sessions Service (to save user sessions) and Google Cloud Storage as an Artifact Service (to save generated files). The processing backend is also written in Python and deployed on GKE.

Challenges during the development

When we started with initial development, we used Gemini 1.5 Pro. Written in KNIME Agent details, KNIME workflows can contain dozens or even hundreds of nodes. When the workflow contains a small number of nodes, it's no problem to provide data for complete workflow and LLM converts workflow without issues.

However when the workflow is bigger, LLM hallucinates. Another thing is that for big workflows, results can also be big and the generated SQL output often exceeded the Gemini 1.5 Pro's token output limit, leading to truncated and incomplete conversions. To overcome these issues, we tried different approaches.

The first one was doing Node by Node conversion. We assumed -and our tests confirmed- that providing for conversion just data for just one Node isn't sufficient, since workflow context, namely predecessor Nodes information is lost. After each Node was converted and we put the results together, it often led to logically inconsistent or semantically incorrect SQL, failing to preserve the intended data flow and transformations, since Nodes usually share context, i.e. they are dependent on each other. This is even more visible when a Node takes results of multiple Nodes as an input.

To overcome this issue, we experimented with providing the right amount of information for LLM and staying with the output token limit. We came up with a deterministic solution that is based on the topological order, and grouping Nodes into groups based on the number of their predecessors and successors Nodes. This turned out to work much better, since we are providing LLM with a contextually rich yet manageable input, thereby mitigating hallucination and improving conversion accuracy.

From the start we were focused on Nodes that are related to Data Engineering, like loading data, transformations, queries, etc. Because of this, the tool doesn't convert all nodes but instead LLM writes a comment that it was not able to convert a Node with a specific id.

To speed up overall conversion, we implemented a parallelization strategy. This involves analyzing the workflow's dependency graph to identify independent branches or sequential groups of nodes whose conversion requests to the LLM could be executed concurrently without requiring outputs from preceding LLM calls.

With the release of Gemini 2.5 Pro we saw a big impact of increased output token limit (from 8192 tokens for Gemini 1.5 Pro to 65536 for Gemini 2.5 Pro). Our tests showed that Gemini 2.5 Pro has good conversion results even when we pass configuration for the whole KNIME workflow. This approach is now our primary conversion method since it's faster. In instances

where this complete conversion fails, for example, owing to the exceptionally complex workflows that still exceed the increased output token limits, the process fails back to the more granular, ‘grouping nodes’ algorithm. While potentially slower, this fallback mechanism ensures robustness and a higher likelihood of successful conversion for challenging cases.

3.2 Big Data stack (e.g Cloudera) application packages

Problem Statement:

Modernizing Cloudera-based data applications into Dataform-powered, cloud-native data pipelines requires more than straightforward code translation. It involves a series of analytical and transformation steps that collectively reconstruct the legacy logic into a scalable, maintainable, and efficient cloud-native workflow.

The migration process begins with a deep analysis of the legacy application packages, which often contain intricate dependencies, diverse technologies, and tightly coupled data-processing logic. Each legacy component must be examined, understood, and replaced with a cloud-native equivalent, similar to solving a complex puzzle, where each piece must fit precisely to create a functional end-to-end data pipeline.

To achieve this, the migration process follows a structured, multi-phase workflow that can be efficiently managed through a multi-agent system, where each agent specializes in a specific task. The high-level stages include:

1. Entry Point Identification:

   Detect the primary execution point or control flow initiator of the application to establish the logical starting node for analysis.

2. Dependency Graph Construction:

   Analyze code interconnections to create a comprehensive dependency graph capturing all data flows, transformations, and relationships between components.

3. Code Traversal and Logic Extraction:

   Iterate through all files contributing to the business logic, capturing key functions, transformations, and configurations relevant to migration.

4. Migration Planning and Target Stack Selection:

   Formulate a detailed migration strategy, mapping legacy components to equivalent Dataform or other cloud-native constructs (e.g. BigQuery, Cloud Composer).

5. Code Conversion:

   Translate legacy scripts (Hive, Impala, Spark, etc.) into the chosen target technology while preserving functional equivalence and optimizing for cloud performance.

6. Pipeline Assembly and Packaging:

   Combine converted code blocks into cohesive, orchestrated pipeline packages compatible with the target environment's execution model.

7. Testing and Validation:

   Validate the migrated pipelines through functional, integration, and regression testing to ensure accuracy, consistency, and data integrity.

8. Deployment to Cloud Infrastructure:

   Deploy the verified pipelines into the designated cloud environment using CI/CD pipelines, Infrastructure-as-Code (IaC), and environment-specific configurations.

This multi-step modernization journey is inherently complex, requiring coordination among specialized components that perform analysis, planning, conversion, and deployment. By adopting a multi-agent architecture, these tasks can be distributed across intelligent agents, each responsible for a specific domain, resulting in a more automated, scalable, and reliable approach to legacy-to-cloud migration.

Solution:

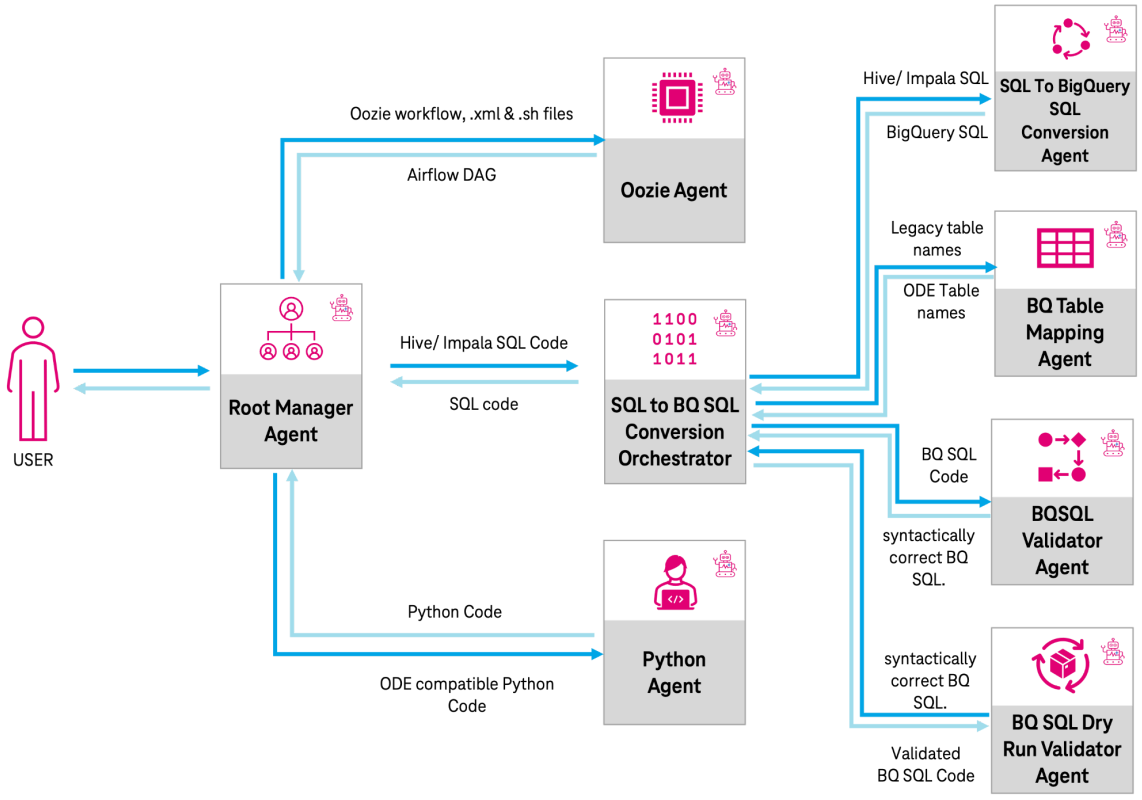

Figure 3: Big Data Code & workflow Conversion using ADK and Gemini

The system has been built using Google's Agent Development Kit (ADK) to implement a modular, agent-based architecture where multiple agents operate together. It uses function tools, agent-as-tools, and Gemini for task coordination and to assign code conversion tasks to the appropriate agent based on file type and context.

1. **Root Manager Agent (RMA)**

   This is the workflow orchestrator for the multi agentic migration/conversion process of Cloudera stack code to equivalent Google Cloud Platform compatible solution. It is the key part of the whole workflow and it is used to communicate with users to gather all the information required to start with. The RMA greets the user and asks for the entry point of the application (for package conversion) or individual file (for standalone conversion). Then RMA analyzes user requests and routes them to the appropriate agent (Oozie, SQL-to-BQ, or Python) for code conversion based on file type.

2. **Oozie Agent**

   The Oozie agent specializes in converting the OOZIE workflows into equivalent airflow DAG without losing the references from supporting files.

3. **SQL to BigQuery SQL Agent**

   In the Cloudera stack most of the queries are written in Hive SQL or Impala SQL dialect. This agent not only specializes in making this code compatible with Google BigQuery, but also in replacing/mapping legacy table names to table names in BigQuery (table mapping sheet needs to be provided). After creating the code, it is able to test syntax against the BigQuery environment and then take corrective actions if needed. These converted scripts are then referred to by the Airflow DAG created by the Oozie Agent.

4. **Python Agent**

   In legacy Cloudera Stack there is some Apache Spark code too, written considering limitations of the system/infrastructure at the time. Also some custom packages used/referred to in the code are no longer needed. The Python Agents goes through the code and removes the part which is obsolete. It also takes care of the part where the code is now going to run on cloud infrastructure and changes data references and connectors accordingly. The final PySpark code is then referred to by the Airflow DAG created by the Oozie Agent.

3.3 PL/SQL to BQ-SQL Migration

Problem Statement:

As established in earlier sections, code conversion is not limited to syntactic translation; rather, it represents only the initial phase of a much more intricate modernization journey. This

complexity becomes particularly evident when migrating applications from Oracle PL/SQL to Google BigQuery.

PL/SQL introduces architectural and operational features that pose unique challenges during migration. The language extensively uses cursors, pointers, user-defined types, and procedural constructs, along with functions and stored procedures invoked directly within SQL queries. These capabilities enable tight coupling between procedural logic and SQL execution, often resulting in deeply embedded business logic within database layers.

BigQuery, in contrast, is engineered around a distributed, analytical, and declarative processing model. While it supports stored procedures and user-defined functions, its execution paradigm emphasizes:

- Set-based query processing
- Stateless operations
- Separation of transformation logic from procedural orchestration
- Parallel, distributed execution rather than cursor-driven iteration

This fundamental difference means that direct syntactic translation is insufficient. PL/SQL structures must be re-architected to align with BigQuery's cloud-native paradigms, requiring thoughtful refactoring of logic, data flows, and control structures.

To bridge this gap effectively, the migration approach must:

1. Analyze procedural dependencies and data access patterns
2. Break down monolithic PL/SQL logic into modular SQL and orchestration components
3. Replace cursor-based logic with set-based transformations
4. Re-implement business logic using BigQuery SQL, scripting, and UDFs where appropriate
5. Externalize orchestration using Dataform, Composer, or similar workflow engines

Thus, migrating PL/SQL workloads is not a simple language transformation exercise; it is a paradigm shift that requires architectural redesign and workflow restructuring to fully leverage BigQuery's scale and performance characteristics.

Solution:

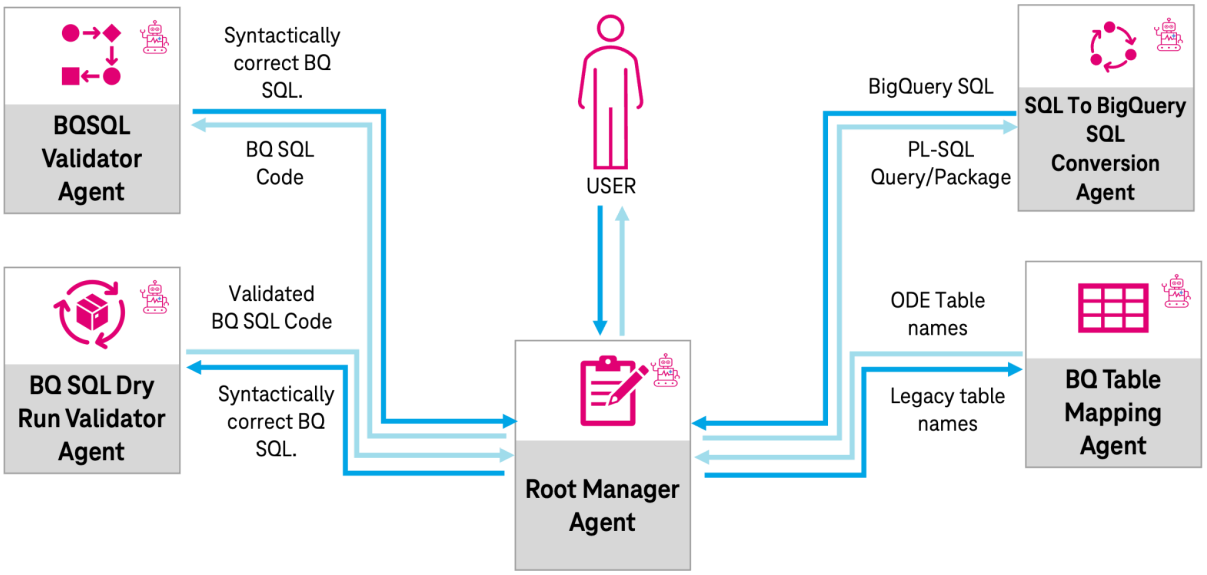

Figure 4: PL/SQL Conversion using ADK and Gemini

Similar to the migration system developed for Cloudera-based workloads, this solution leverages Google's Agent Development Kit (ADK) to build a modular and scalable multi-agent architecture. The system employs function-based tools, agent-as-tools patterns, and Google's Gemini models to orchestrate tasks, route work dynamically, and execute specialized code-conversion functions based on file structure, data context, and transformation requirements.

This architecture ensures distributed ownership of tasks, parallel processing where appropriate, and human-in-the-loop capability for high-accuracy enterprise migrations.

1. **Root Manager Agent (RMA)**

   Role: Workflow Orchestrator & User Interaction Interface.

   The RMA serves as the primary coordinator for the PL/SQL to BigQuery SQL migration lifecycle. It initiates user interaction, gathers required inputs, and determines the migration scope whether converting a single PL/SQL script or an entire package.

   Key responsibilities include:

   - Greeting the user and collecting application entry points or file references
   - Understanding migration intent (package-level or isolated file conversion)
   - Routing tasks to specialized agents based on file type and context
   - Managing progress flow and ensuring continuity across tasks
   - Maintaining overall state and orchestrating inter-agent communication

The RMA ensures that user inputs, code assets, and context are consistently propagated throughout the workflow.

2. SQL to BigQuery SQL Conversion Agent

Role: Core Code Translator & Logic Reconstruction Engine.

This agent performs semantic and syntactic translation of PL/SQL logic to BigQuery SQL, ensuring functional equivalence while adapting to differences in execution models.

Key responsibilities:

- Understanding full PL/SQL package logic, including nested structures
- Rewriting SQL using BigQuery compatible syntax and functions
- Identifying pointers, cursors, procedural logic, or unsupported constructs
- Designing equivalent BigQuery patterns (set-based rewrites, SQL scripts, UDFs)
- Producing clean, optimized BigQuery SQL as output

This agent does more than syntax conversion; it performs logic refactoring to align with BigQuery's distributed architecture.

3. BQ Table Mapping Agent

Role: Dataset & Table Reference Transformation.

This agent extracts all source table references from the PL/SQL code and maps them to corresponding BigQuery dataset and table identifiers.

Key responsibilities:

- Parsing all table references from PL/SQL files
- Mapping source schema elements to BigQuery datasets and tables
- Engaging the user where mappings are unknown (human-in-the-loop)
- Persisting mapping knowledge to a reusable lookup store
- Automatically replacing legacy names with cloud-native equivalents

This ensures consistency, avoids manual lookup effort, and builds an evolving schema intelligence layer for future migrations.

4. BQ-SQL Validator Agent

Role: Quality Assurance & Knowledge-Accumulating Validator.

This agent validates the converted BigQuery SQL and maintains a validation knowledge base using Retrieval-Augmented Generation (RAG).

Key responsibilities:

- Detecting structural, functional, and logical inconsistencies
- Suggesting and applying corrections
- Documenting issues and resolutions into a validation RAG store
- Ensuring previous mistakes do not recur
- Verifying conversion accuracy across multiple iterations

This agent acts as the continuous improvement and QA loop, enhancing accuracy over time.

5. BQ-SQL Dry Run Validator Agent

Role: Runtime Compatibility & Type-Safety Checker.

To ensure correctness at execution time, this agent performs Dry Run validations against the target BigQuery environment.

Key responsibilities:

- Executing dry-run queries without consuming compute
- Identifying schema mismatches, missing columns, and type incompatibilities
- Correcting or recommending adjustments for failed queries
- Ensuring final query is executable in the production BigQuery environment

This final validation ensures the generated pipeline can run safely in real workloads even when full metadata context is not initially available.

Appendix: PL/SQL to BQ-SQL (previous approaches before Gemini 2.5 Pro)

The model architecture for PL/SQL to BQ-SQL code conversion tool is designed around a sequential pipeline of Directed Acyclic Graphs (DAGs), each performing a specific role in the conversion process. This architecture leverages Large Language Models (LLMs) within Google Cloud Platform's (GCP) Cloud Composer environment, which provides the scalability and flexibility necessary for handling complex code conversion tasks.

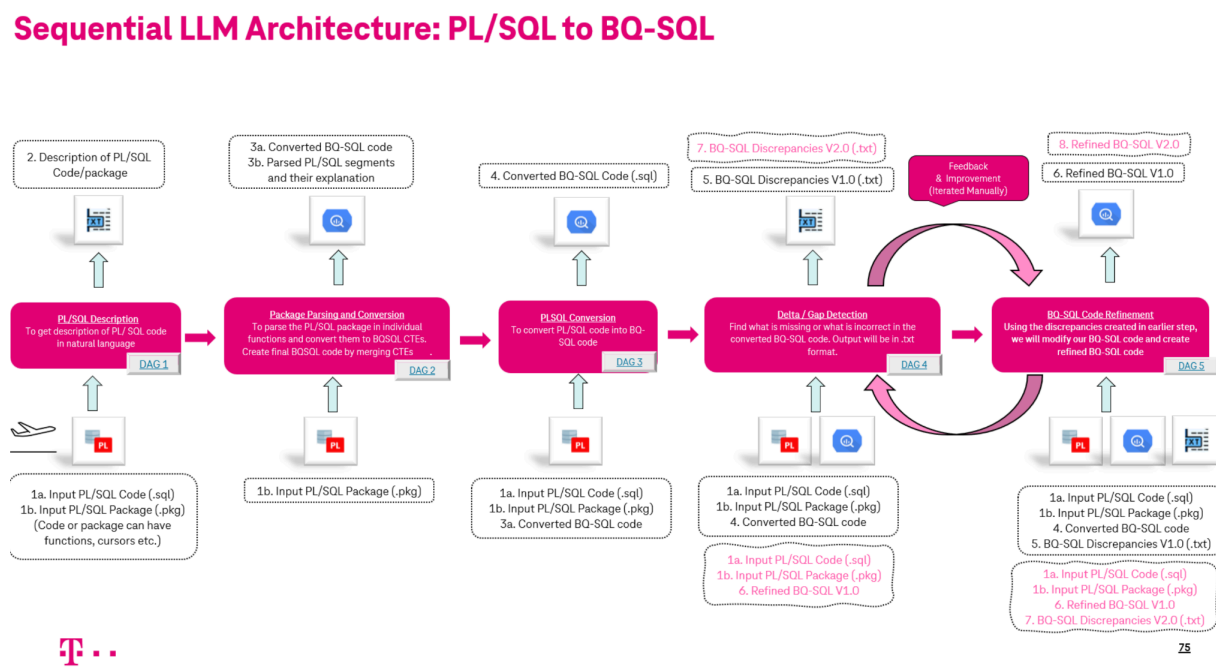

Figure 5: PL/SQL to BQ-SQL Code Conversion using Gemini 1.5 Pro

1. DAG 1: PL/SQL Description Generation

The process begins with the generation of detailed natural language descriptions of the input PL/SQL code, including its functions, procedures, and dependencies. This step is crucial for understanding the code's structure and preparing it for subsequent conversion steps.

2. DAG 2: Package Parsing and Conversion

Once the code description is generated, the next step involves parsing PL/SQL packages into individual functions. These functions are then converted into BigQuery SQL (BQ-SQL) Common Table Expressions (CTEs), forming the building blocks for the final BQ-SQL code.

3. DAG 3: PL/SQL to BQ-SQL Conversion

This DAG takes the parsed PL/SQL components and performs the core conversion into BQ SQL code. The LLM handles both .sql and .pkg files, ensuring that all elements of the PL/SQL code are accurately translated into the target SQL language.

4. DAG 4: Discrepancy Detection

After conversion, the tool compares the newly generated BQ-SQL code with the original PL/SQL code. This comparison focuses on identifying discrepancies in syntax, logic, and functionality, which are then documented in detailed reports.

5. DAG 5: BQ-SQL Code Refinement

Based on the discrepancy reports, the BQ-SQL code is refined to better align with the original PL/SQL logic. This iterative refinement ensures that the converted code meets the required standards of accuracy and performance.

Each DAG in this architecture is parameterized, allowing users to specify input and output paths, as well as the LLM used for each stage of the process. While the tool supports various LLMs available on GCP, the preferred model is Gemini 1.5 Pro, which has demonstrated superior results in handling complex code conversion tasks. This sequential, iterative architecture not only streamlines the conversion process but also allows for continuous improvement through manual intervention and feedback loops at each stage.

References

[1] Google Cloud Platform. Oracle to BigQuery SQL Translation. Google Cloud Documentation, 2024. Available: https://cloud.google.com/bigquery/docs/migration/oracle-sql

[2] Google Cloud Platform. Automated SQL Translation for Data Warehouse Migrations. Google Cloud Blog, 2024. Available: https://cloud.google.com/blog/products/data-analytics/automated sql-translation

[3] Google Cloud Platform. Batch SQL Translator for Migration to BigQuery. Google Cloud Documentation, 2024. Available: https://cloud.google.com/bigquery/docs/batch-sql-translator

[4] Google Cloud Platform. Introduction to BigQuery SQL. Google Cloud Documentation, 2024. Available: https://cloud.google.com/bigquery/docs/reference/standard-sql

[5] Dataform. Google Cloud Documentation, 2025. Available: https://docs.cloud.google.com/dataform/docs/overview

[6] Google Agent Development Kit, 2025. Available: https://google.github.io/adk-docs/

[7] Gemini 2.5, 2025. Google Cloud Documentation. Available: https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/, https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro

[8] KNIME Analytics Platform, 2025. Available: https://www.knime.com/software-overview

[9] MostlyAI Synthetic Data, 2025. Available: https://mostly.ai/synthetic-data-basics

[10] Cloudera, 2025. Available: https://www.cloudera.com/

[11] Apache Hive, 2025. Available: https://hive.apache.org/

[12] Apache Impala, 2025. Available: https://impala.apache.org/

[13] Apache Oozie, 2025. Available: https://oozie.apache.org/

[14] Oracle PS/SQL, 2025. Available: https://docs.oracle.com/en/database/oracle/oracle-database/18/lnpls/index.html

[15] CrewAI. Overview of Multi-Agent Frameworks for AI Workflows. CrewAI Documentation, 2024. Available: https://crewai.io/docs

[16] AutoGen. Introduction to AutoGen Multi-Agent Framework. AutoGen Documentation, 2024. Available: https://autogen.ai/docs