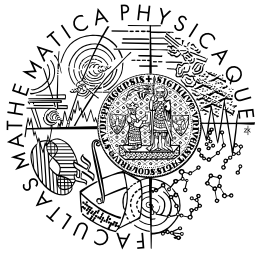

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Jurčo

Data Lineage Analysis Service for Embedded Code

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2023

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), I hereby grant the so-called MIT License. The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and / or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

Copyright (c) 2023 Michal Jurčo
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

In Prague date ....

Author's signature

This way I would like to thank my supervisor, doc. RNDr. Pavel Parízek, for his guidance, help and motivation while working on this thesis.
Thanks should also go to RNDr. Lukáš Hermann for his invaluable assistance regarding MANTA Flow platform and mentorship.
I am especially grateful to my fiancée, Kristýna, who has shown her never-ending support and love throughout my studies that helped me get this far.
Last but not least, I want to thank my family and my cats for always providing comfort when I needed it.

Title: Data Lineage Analysis Service for Embedded Code

Author: Michal Jurčo

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data integration tools often use embedded code for data manipulation tasks. Popular examples of such tools include AWS Glue data integration service, Databricks platform, Snowflake data cloud or SQL Server Integration Services (SSIS). Embedded code is typically written in programming languages such as Python, Java, C# or JavaScript. Manta Flow is an automated platform that can analyze data lineage in database models, data pipelines of data integration tools, and in application source code, but it lacks the ability to analyze embedded code. In this work, we discussed potential ways to extend the capabilities of Manta Flow with the ability to analyze data lineage in embedded code. We created a general design of a reusable Embedded Code Service that leverages the existing potential of data flow analysis of source code, and uses it to analyze embedded code. We implemented a specialization of this service for the Python programming language, and to demonstrate its usefulness, we designed and implemented a prototype of data lineage scanner for AWS Glue data integration service. This scanner extensively uses the service to analyze data lineage in embedded Python scripts, which we demonstrated on a realistic example.

Keywords: data lineage data flow embedded code python AWS Glue

Contents

1. Introduction

Prior to the advent of computers, data processing was predominantly a manual task that involved a considerable amount of time, effort, and the potential for human errors. Organizations, particularly businesses, faced challenges in handling large volumes of data efficiently and accurately. This led to a demand for automated systems that could streamline data processing and eliminate the limitations associated with manual methods. The development of computers offered a solution to these challenges by providing a means to automate data-related tasks. It allowed storing and processing larger amounts of data than ever before.

A couple of decades later, data is one of the most important resources for many companies. It helps them run their business more efficiently and make better business decisions. Data pipelines have emerged as crucial components in modern business infrastructures. In general, a data pipeline refers to a system or framework that facilitates the flow of data from various sources to its destination, typically for processing, analysis, storage, or visualization. It is a series of interconnected steps or processes that enable the extraction, transformation, and loading (ETL) of data, ensuring that it moves efficiently and reliably through the pipeline.

A simple example of a data pipeline might be saving contents of a form filled by a customer on a web page to a database. Another, a more complex one might be a preparation of marketing success report where data from recent sales stored in an accounting system are correlated with a list of latest advertising campaign exported from a marketing tool and compared with customer satisfaction form result which might be stored in a database as in the previous example. It is easy to see that these pipelines might become long and complex and that they can be chained one after the other. Less obvious is that they are often facilitated by multiple systems. It would be easier to manage if they were facilitated by one, but there are often reasons why that is not possible. There is no universal tool, each serves a different purpose. A database is great for storing data and fast queries over large data sets, ETL tools are good in data transformations, reporting tools are great for data visualization and analytic and machine learning tools are essential for data science. There might even be a legacy system that cannot be easily migrated to a different platform. They all compose a data environment which serves one purpose - to run business more effectively and make good business decisions.

As the Greek philosopher Heraclitus said, *the only constant in life is change*. This statement is fitting for data environments, because they always change. A new column is added to a schema, two are removed, a new process is introduced or an existing one needs to be modified or fixed etc. A change may span across multiple systems or a modification in one of them may influence others that depend on it. Planning it can become a nightmare, because even if there is a support for impact analysis in each system, there is none environment-wide and has to be prepared manually. That is why additional processes were developed that help making changes with predictable outputs, without causing errors and that ensure data correctness afterwards. One of such processes is called *data lineage*.

1.1 Data lineage

Data lineage is a process of mapping and visualizing data flows within a data environment. It tracks data as they flow from various sources to their destinations and their transformations with aim to help manage and develop data environments. It provides a comprehensive understanding of how data is acquired, manipulated, and utilized within an organization. Data lineage offers several benefits to businesses and data professionals. Firstly, it enhances data governance and regulatory compliance by ensuring transparency and traceability of data. It enables organizations to meet the requirements of various regulations such as GDPR or CCPA. Secondly, data lineage improves data quality and accuracy by identifying data inconsistencies, errors, or gaps in the data flow. This helps in identifying and rectifying issues promptly, leading to reliable and trustworthy data insights. Additionally, data lineage facilitates data discovery, data integration, and data analytics processes, as it provides a clear understanding of data origins and transformations. It enables faster troubleshooting and root cause analysis, reducing the time and effort required for resolving data-related issues. Overall, data lineage plays a crucial role in maximizing the value of data assets and ensuring data integrity, trust, and accountability within an organization.

Data lineage can be obtained by hand from teams of analysts that map data environments or, more recently, using one of the automated systems that are being developed. Manual data lineage analysis is time- and labor-intensive, so developing automated solutions can provide more up-to-date results and decrease costs.

Manta Flow is an automated data lineage platform. It can analyze complex data environments consisting of various databases, data integration and reporting tools and applications in Java, C# or Python. Using metadata extracted from each connection to a system, Manta Flow computes data lineage graphs where vertices represent data sources and directed edges represent data flows between them. At first, a graph is constructed and stored for each individual connection. When the data lineage is visualized, the graphs are retrieved from the repository and combined together to show a graph of the entire environment.

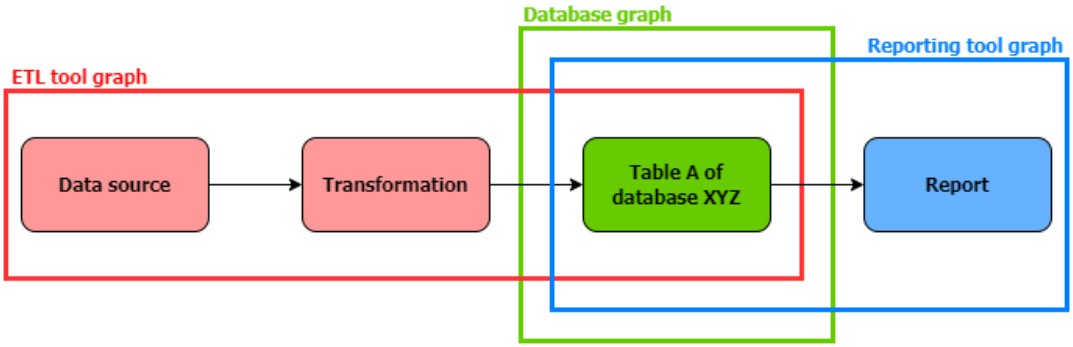

Figure 1.1: An example of a combined data lineage graph

To illustrate this with an example, let us have an ETL tool which contains a pipeline writing data into table A of database XYZ and a reporting tool which visualizes data from this table in a report. The analysis would produce three

graphs:

- an ETL tool graph which contains data pipeline data flow ending with a write to table A,
- a database XYZ graph which represents table A and its columns,
- a reporting tool graph that contains data read from table A into a report.

These graphs would be visualized as one unified data lineage, as seen in Figure 1.1.

1.2 Embedded code

Various data processing, management and analytic tools have been developed to allow their users to extract important pieces of information from the vast amounts of data they collect. These tools often provide graphical interface for creating data pipelines consisting of commonly used data sources and transformations. Using such tool opens data pipeline management to a wider, less technically proficient audience, so business-oriented employees can be involved more closely in the development. An example of such tool may be AWS Glue platform where users may define their ETL pipeline using graphical interface with multiple data sources, transformations and joins without having to write a single line of code. However, data can be very dynamic and different and it would be difficult to define every possible data transformation that the users might require. In such cases these tools often allow extending the pipeline with embedded code.

Embedded code is a piece of code provided by the user that is safely executed in the context of the tool and can perform (almost) any desired task. It is often a function or a script. Popular examples of tools that support embedded code include already-mentioned AWS Glue, but also Databricks platform, Snowflake data cloud or SQL Server Integration Services (SSIS) etc. Programming languages used in embedded code often include the popular and well-known ones such as Python, Java, Scala or C#.

Embedded code can be used to perform a data transformation in an ETL pipeline that is not included in the standard toolbox, read data from an unsupported data source or to create a user-defined database function that cannot be written efficiently in SQL.

Data flow analysis of embedded code is a crucial missing link in Manta Flow. It can analyze both data pipeline metadata and standalone Python, Java or C# applications, but there is no support when such a piece of code is a part of the pipeline. Missing embedded code data lineage causes logical gaps in the holistic data lineage and decreases its usability, because these gaps have to be filled in by hand. Following the recent surge in demand for data science and machine learning solutions where especially Python is the programming language of choice, the number of enterprises that use embedded code in their data environment has also risen significantly. This drives the market demand for data lineage solutions that can cope with it. As there are currently no solutions that can reliably provide such data lineage, extending the current capabilities of Manta Flow to analyze data lineage in application source code with the ability to also analyze embedded code shall provide it a significant competitive advantage.

1.3 AWS Glue

AWS Glue is a serverless data integration service that makes it easier to discover, prepare, move, and integrate data from multiple sources for analytics, machine learning and application development. It performs data processing on Apache Spark engine in a cloud environment. Data pipelines are defined using embedded code, supported programming languages are Python and Scala. They can also be created from the GUI, where the tool generates the corresponding pipeline code. This service is especially convenient for companies that already use other AWS services as it can efficiently use such resources.

AWS Glue has been on top of the list of technologies to be supported by Manta Flow based on customer enquiries. There is currently very limited support for automated AWS Glue data lineage on the market, so offering it provides a competitive advantage. As the pipelines consist of embedded code, analyzing it is the best, although a very difficult way to extract data lineage information. However, Manta Flow can already analyze Python and Bytecode (Scala is compiled into Bytecode) applications. Providing support for embedded code analysis is therefore a direct prerequisite for a successful AWS Glue data lineage solution.

1.4 Goals

The goal of this thesis is to design a data lineage analysis service for embedded code in Manta Flow that will enable integration of data lineage graph from data processing and analytic tools with the data lineage graph derived from the embedded code.

One of the main tasks is to create a solid design of the service that should be easily extendable with support for new tools and their embedded code in the future. Benefits and usefulness of this design will be then demonstrated on a prototype implementation of the service for AWS Glue and the embedded code written in Python.

Other specific tasks include a proof-of-concept implementation of a metadata extractor for AWS Glue and modifications to the existing Python scanner. A very important aspect of the service is high performance, because it will be called many times during a run of the Manta Flow analysis platform, specifically every time the analysis processes a statement that executes a piece of embedded code.

1.5 Glossary

Let us define a few important terms that are often used in this work.

- *Data flow* refers to the path and direction of data movement, capturing the sequence of transformations and processing steps that data undergoes throughout its lifecycle.
- *Data lineage* is the record of the origin, transformations, and movement of data, providing a clear and traceable path from its source through various processes and systems to its destination.
- *Manta Flow* is a state-ot-the-art automated data lineage analysis platform.

- We will use the term *data technology* to uniformly reference databases, ETL and reporting tools. This term shall simplify naming all systems, frameworks, platforms and tools that can be used for data processing and management in the work.
- *Manta scanner* is a component of Manta Flow specialized to perform data lineage analysis for one data technology.
- *Embedded code* is a piece of code (a script, a class, a package) embedded in a data technology that is executed in a specific runtime environment to perform a specific task.
- *Metadata* refers to descriptive and contextual information about data, encompassing details such as data source, format, structure, meaning, and other attributes.
- In the context of embedded code analysis, *source technology* is the data technology that uses embedded code, or from a different point of view, a data technology that is the source of embedded code.
- *Embedded Code Service* is a service for data lineage analysis of embedded code.

1.6 Outline

This thesis is split into several chapters. In introduction we briefly describe the motivation and the goal of the thesis. In the second chapter we describe important aspects of MANTA Flow platform with which the service developed in this thesis is integrated in. Chapter 3 is dedicated to a detailed problem analysis and we formulate our requirements there. Chapter 4 delves into design and implementation of embedded code service and changes that were made to Python scanner. In chapter 5 we take a look at the design and proof-of-concept implementation of AWS Glue scanner, which uses Embedded Code Service for data flow analysis of embedded Python code. In chapter 6 we demonstrate the functionality of implemented features on several examples and discuss the limitations of the implementation. Finally, in conclusion we sum up what we achieved in this work and how we did it.

2. Manta Flow platform

Before we get into the details of data lineage analysis service for embedded code, let us first introduce and describe Manta Flow platform. It is the platform that the service is integrated with and influences many of the design and implementation decisions. It also gives some examples of solutions to similar problems we will encounter, which we will use as an inspiration in our solutions.

2.1 Manta Flow overview

Manta Flow is an automated data lineage platform that scans data environment to build and visualize a graph of all data flows (as seen in Figure 2.1) within it. This graph enables its users to get better visibility and control of their data processes. The emphasis is on *automation*, Manta Flow requires little intervention apart from configuration and can perform data lineage analysis in a matter of hours to days, compared to manual analysis which can take weeks to months, so the visualization provides up-to-date information.

Data lineage analysis is based on the analysis of metadata and scripts extracted from the connected systems. Metadata contain information about schema and structure of internal entities and their relations. Scripts contain data processing and transformation logic applied on these entities. It is not possible to construct complete data lineage from just one of these sources.

Manta Flow is made up of three major components.

- *Manta Flow CLI* is a Java command line application that extracts all metadata and scripts from source data technologies, analyzes them, sends all gathered metadata to the *Manta Flow Server*, and optionally, processes and uploads the generated export to a target metadata database.
- *Manta Flow Server* is a Java server application that stores all gathered metadata inside its metadata repository, transforms it to a form suitable for import to a target metadata database, and provides it to its visualization or third-party applications via API.
- *Manta Admin UI* is a Java server application providing a graphical and programming interface for installation, configuration, updating, and overall maintenance of Manta Flow.

Each data environment consists of a different set of data technologies. Each data technology stores its internal data in a different metadata structure and provides a different API to access it. To be able to analyze the entire environment, Manta Flow CLI uses a wide range of proprietary scanners, currently there are over 40. Each scanner can connect to and analyze data lineage of a single data technology to accommodate its specific structure. It produces a common metadata output uploaded to Manta Flow Server's repository. This architecture allows an easier support of a new data technology by developing a new scanner.

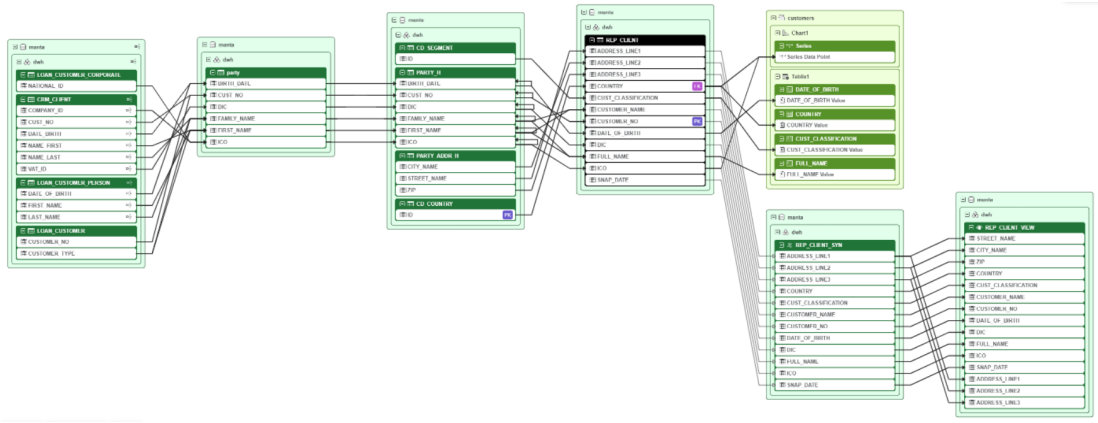

Figure 2.1: An example of data lineage visualization in Manta Flow

2.2 Manta graph

The common output of all scanners is a data structure called *Manta graph*. It allows to capture both data environment structure and data flows within it. A graph consists of nodes (vertices) and edges between them. A node may represent a data source or a transformation, e.g., a database table column, a report dimension or a union of columns. Edges are directed and represent data flow from one node to the other.

Nodes have a layered structure to allow representation of a logical hierarchy. This is expressed in a child-parent relationship between nodes, e.g., a server node could be a parent to several database nodes, each database node would have table node children and in each table there would be column node children. However, only the nodes on the lowest level can be connected with data flow edges (e.g. columns). This hierarchy is important for visualization to simplify grouping of related nodes such as all columns of a table and to define node paths.

Each node is identified by a unique path derived from its name and names of its ancestor nodes, which enables merging the same entity and associated data flows in different graphs. For example, if an MSSQL database table is a source of a view in that same database and is also used in a report in SSRS reporting tool, a node representing it will be created with the same identification in MSSQL scanner output and in SSRS scanner output. Manta Flow Server is then able to merge these outputs to visualize one node with two distinct data flows. This can be seen in Figure 2.1 where SSRS nodes are framed in a different shade of green than those belonging to MSSQL.

2.3 Scanners

Scanners are important building blocks of Manta Flow. The role of a scanner is to analyze data lineage in systems of a particular data technology. Ins and outs of each data technology are different, so each scanner does its job in a different way, but they all have a similar structure. They consist of two main components, a *Connector* and a *Dataflow Generator*.

2.3.1 Connector

The purpose of a *Connector* component is to facilitate a connection to the analyzed system and prepare metadata for analysis. That consists of two steps, metadata extraction and creation of a general model from the extracted metadata.

Extractor

Metadata extraction is performed by the *Extractor* sub-component of the Connector. Its goal is to connect to the analyzed system using the configured connection options (URL, credentials or access tokens) and extract all metadata required for data lineage analysis. Metadata can contain information such as database schema, definition of reports with all dimensions and facts that are used in them or enumeration of ETL pipelines with their sources, destinations and individual transformations. Other artifacts could be extracted, too, such as scripts of any kind, configuration etc. All extracted resources are stored locally in one place so that the analysis can be executed in *offline* mode, that is, without requiring an active connection to any of the systems. The format in which they are stored is not important, however most often the format of raw extracted metadata is used.

Database scanners usually go one step further and create a so-called *data dictionary* from the extracted metadata. Data dictionary stores schema of extracted database resources in a universal data structure that is able to accommodate different hierarchical arrangements of databases. It is a kind of pre-processing so this information can be readily used by other scanners in the later stages of the analysis should they need it. Extraction is always executed before other stages of data flow analysis for all scanners.

Reader

The other sub-component of a Connector is called *Reader*. Its purpose is to read extracted metadata and create a general model that can be used for data flow analysis. This model serves as an interface between a Connector and a Dataflow Generator. It helps to create an abstraction of metadata, because its format or the way it is read may change in time.

2.3.2 Dataflow Generator

Dataflow Generator discovers data flows in target systems using the extracted inputs. The output of such analysis is a Manta graph. There are multiple steps in this process and the specifics vary between different scanners. Usually it includes translating structured metadata into nodes and edges following data flow rules. Suppose that we have created a report in a tool such as Tableau. We used a database view as our data source and we visualized one of the columns as a dimension in the report. Furthermore, we computed another dimension from two other columns. Dataflow Generator would create nodes for the view columns, nodes for the dimensions in the report and then add edges between the columns and dimensions in question.

Database scanners also analyze queries and scripts present in databases. There are Data Definition Language (DDL) scripts which define database objects (tables, views, triggers, stored procedures etc.). Analyzing them is more complicated than creating data flows based on metadata. The algorithm for processing database queries and scripts is based on parsing a well-defined language. For example, a database view is created from a database query, so first, the query is resolved, the columns of the tables included in the query are created and then connected with the corresponding view columns.

2.4 Dataflow Query Service

Sometimes, a scanner for one data technology runs into a database query that defines a data source. It is a common feature of many reporting and analytic tools. It would not make sense to include a support for each SQL dialect in all such scanners, firstly because it is quite complicated, but also because there are scanners that can process it already. Furthermore, there might not be enough information in the context of this scanner to resolve such query. Take for example a query `SELECT * FROM TABLE_A`. The scanner doesn't know the columns present in `TABLE_A`, so it is not possible to provide an accurate representation.

Dataflow Query Service is a component that was introduced to help with processing of embedded SQL queries. It provides a unified service interface, which allows launching execution of a database scanner on a small input, that can be used by other data technologies. It works by selecting the appropriate scanner for the provided input, executing data flow analysis and merging the resulting sub-graph into the graph of the original data technology scanner.

One of the main benefits of this service is that it uses data dictionaries created by each scanner in the extraction phase, therefore it has access to extracted database schemas, which allows it to resolve exact columns for queries such as `SELECT * FROM TABLE_A`. It is also useful in situations when details about the target systems are unknown or unresolved, the service has the ability to deduce columns from available information and to choose the appropriate scanner from the provided connection details.

Overview

In general, there are three pieces of information needed to analyze an SQL query:

- Connection data - connection type, connection string, server hostname (at least connection string or server name is needed).
- Default data - default database, default schema, connecting user (as applicable) in case connection data is not available or not recognized.
- Query or embedded script text.

If the data technology scanner has all of the above information, it constructs a `Connection` object directly, otherwise it is expected to perform connection mapping by retrieving the required data from manual configuration. Connection

and query text are inputs for Dataflow Query Service. Next, dictionary mapping is performed, that is, an appropriate target scanner and a persisted data dictionary is selected based on `Connection` data. The selected scanner is invoked with query text, data dictionary and defaults, which provides the result in form of a `DataflowQueryResult`. The data technology then uses this result to connect the result nodes to the relevant nodes in the host graph and to merge these graphs together.

External connections

One of the main purposes of `DataflowQueryResult` is to perform connection to external nodes. The purpose of embedded queries is to provide data for further processing, therefore we would like to connect the nodes of the query objects with other nodes in the lineage, e.g., connect database columns with data fields in a report. These connection points are unknown to the query scanner and the host scanner doesn't understand the query in advance, so the result graph contains extra nodes called *pin* nodes.

Pin nodes represent an input to a node representing query parameter or an output from a query resultset column. After the query is analyzed and its graph is created, the service adds these pin nodes and connects them with appropriate edges to the result nodes. Then, the host scanner might provide a mapping from pin nodes to nodes in the host graph. The service creates new edges from pin nodes to host graph nodes (and vice versa, depending on the direction) based on the provided mapping and then contracts the pin nodes, thus effectively connecting the resultset column node or a parameter node with the host graph node. Should any pin node remain unmatched, it is filtered out in the filtering task later in the process. Pin node mapping is depicted in Figure 2.2.

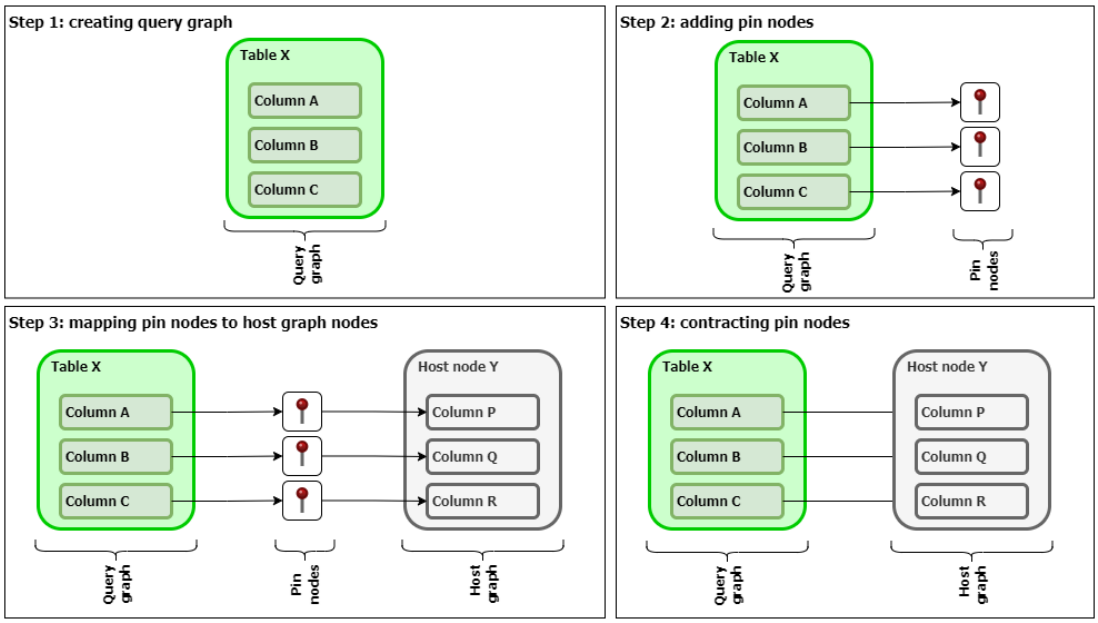

Figure 2.2: A diagram of the pin node mapping process

Architecture

`DataflowQueryService` is a common interface that provides methods for client scanners for analyzing SQL queries or creating hierarchical database structures. This interface is implemented by:

- Individual database query services that contain logic for providing data flows or hierarchical database structures.
- Proxy class called `DataflowQueryServiceImpl` that wraps all other individual query services. It is a classic implementation of a proxy class that chooses a specific query service based on information provided in the connection from the caller and delegates the operation.

A simplified class diagram is depicted in Figure 2.3.

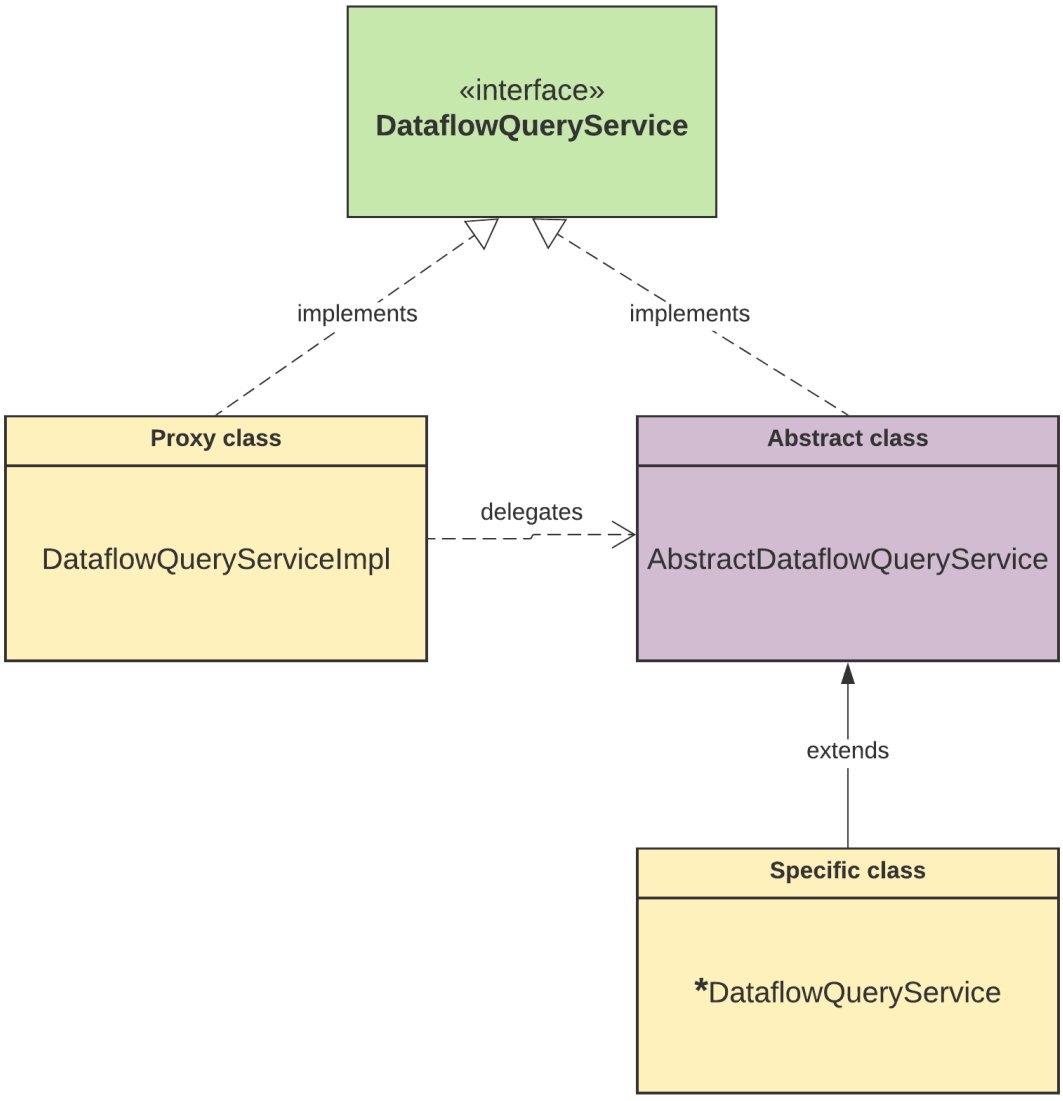

Figure 2.3: A simplified class diagram of Dataflow Query Service

2.5 Programming Language Scanners

It might seem that analyzing all databases, ETL, reporting and analytic tools could yield a complete data lineage of an environment, but the fact is that these are not all data technologies that take part in data pipelines. There are also other applications, often developed by the companies themselves that engage in data processing. As these applications are unique, the only thing they have in common with other applications is the programming language used to write the source code. Therefore, the only universal way to analyze them is to perform the data flow analysis of the source code.

Programming language scanners are one of the most complex scanners in Manta Flow. There are currently three of them: Bytecode scanner, C# scanner and Python scanner. The scope of processes that can be expressed in a programming language is much greater and more complex compared to other data technologies. These scanners perform static code analysis to analyze data flows in the application and construct data lineage graph.

Static code analysis is a method of analyzing source code without executing it. It involves examining the code's structure, syntax, and logic to observe certain properties. Often it is used to improve code quality and detect issues that could lead to runtime errors or security vulnerabilities, but it can have many other use-cases.

There are other ways to analyze the code, but they are not suitable in the situations where Manta Flow is used. We cannot run the code for the analysis purposes, because it causes side effects (the pipeline would be executed). We also don't want to force the customers to modify the source code to record the events that are happening for the purposes of data lineage analysis, because often there are hundreds to thousands of code files that they want to analyze. It would be very labour intensive and minimizing labour is one of the reasons they use Manta Flow. We can only look at the code, so static analysis remains as the most suitable option.

The goal of this static analysis is to find where data is read into the application, track it across all transformations and then find where it is written out. This proved to be a difficult task. There are several approaches for static code analysis, but none of them is aimed at data lineage, so a custom algorithm had to be developed. It uses symbolic analysis and an iterative approach along with multiple optimisations to produce limited results in a limited environment. These limitations are computing power, memory space and time. Manta Flow is expected to run in a common enterprise environment on a machine with a standard multi-core CPU, a reasonable RAM size (e.g. 32GB). Furthermore, the analysis is expected to end in the span of hours up to a few days. Due to that the output focuses on visualizing data reads and writes and data flows between them but it does not show details of intermediate transformations.

2.5.1 Data flow analysis of source code

Let us describe how this analysis works in more detail. It will help us understand problems and solutions later in this work. All programming language scanners follow a similar workflow, but they implement each step in a little dif-

ferent way due to the differences between the languages. Let us focus more on the details of Python scanner as it will be more relevant for the rest of this work. The structure of programming language scanners is also a bit different from other scanners, the core of the data flow analysis is performed by the Reader component as opposed to Dataflow Generator in other scanners. We shall explain why later in this section when more context is provided. A workflow diagram can be seen in Figure 2.4.

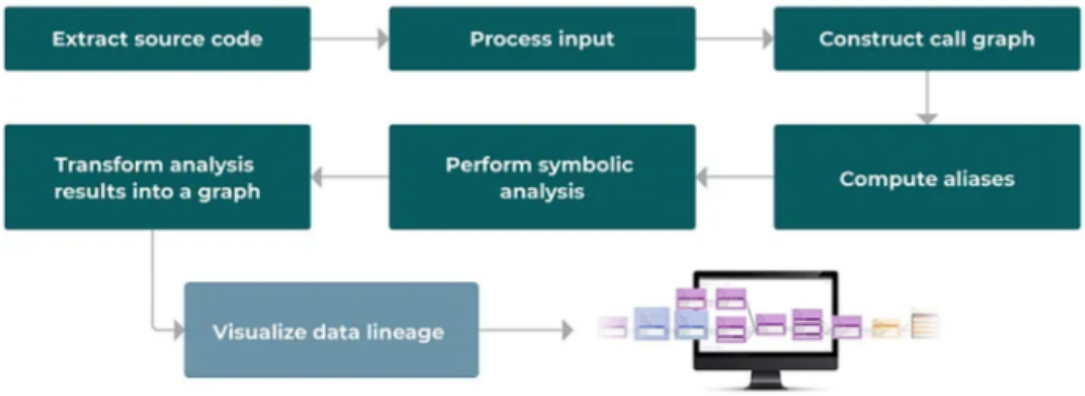

Figure 2.4: A simplified diagram of Python scanner workflow

Let us use the following simple Python program called `json_to_csv.py` shown in Figure 2.5 as an example. The program converts a JSON file to CSV format. Firstly, a JSON file is read using a built-in Python library `json`. Then, using a popular Python library for data manipulation called `pandas`, the contents of a JSON file are converted to a data frame, which is an abstraction of tabular data. Finally, the data frame is written to a file in CSV format. Clearly, the data flow in this program is pointing from the JSON file to the CSV file. Let us now take a look what at happens inside the Python scanner to this program.

```
1  import pandas as pd
2  import json
3
4  # Loads json from file
5  def get_customer_data():
6      with open("customer_data.json") as file:
7          return json.load(file)
8
9  customer_data = get_customer_data()
10 df_customer = pd.json_normalize(customer_data)
11
12 df_customer.to_csv("customer_data.csv")
```

Figure 2.5: Sample Python program json_to_csv.py

Source code extraction

The first step is source code extraction. Before the analysis can begin, all inputs need to be collected and arranged in the form they are expected to be in.

The applications usually consist of the application code and of external libraries. The user only needs to provide the source code they wrote (so in our example only `json_to_csv.py`). The code of supported 3rd party libraries (`json` or `pandas`) that are used is added by the scanner as it stays the same across different use cases. When the files are collected and arranged, an extraction configuration is generated which captures the structure of the input and allows the user to specify which functions or modules should be considered as analysis entry point.

Analysis entry point is the routine that shall be considered as a starting point of a program execution. Usually, it can be the `main` method in Java or C# or the module containing `__main__` block in Python, but in general it can also be any other function, method or module. This defines the starting point of the analysis, and from that point, function invocations and variable assignments are tracked. In our example, it would be the `json_to_csv.py` module.

Source code extraction is a standalone step and represents metadata extraction for programming language scanners. It is a part of Extractor component.

Input processing

Now we are moving to Reader component of Python scanner. In this initial step of data flow analysis, the extracted inputs are read from file system and pre-processed. In Python, this involves parsing of the source code into a convenient internal representation. There are a few data structures that need to be created in this step because they are used in the following ones.

One of these structures is class hierarchy, which helps with the detection of invocation targets (this is implemented in Bytecode and C# scanners, but not yet in Python scanner). The other, more important for the algorithm is *call graph*. It captures caller-callee relationship between executables (functions, methods, modules) in the application. A caller is the executable that invokes another executable in its body, a callee is the executable that is being invoked by another executable. Call graph helps to find which other executables need to be analyzed again after the analysis of one, because its result might influence the callers and callees. This will be explained in more detail later.

Alias analysis

Aliases are different expressions that might reference the same value. It is important to analyze assignments in the application to resolve these aliases, because a value assigned into one of these expressions has to be propagated into all aliases. Similarly, if a value in an expression is modified, so it is in all of the aliases. An alias could be understood as a synonym to a reference, but it tracks a few more cases than just references. Figure 2.6 shows how an alias can be created.

```
1 foo = "Hello World!
2 bar = foo # bar now aliases the same value as foo does
```

Figure 2.6: An example of an alias

Symbolic analysis

With all the preparatory steps done, we now have everything ready for *symbolic analysis*, which is the core of the data flow analysis. It is based on an iterative analysis of executables and propagating data flows between symbolic expressions as defined by assignments and function invocations.

Data flows (or shortly just *flows*) represent a meaningful data lineage information in the source code. We can split flows into two groups. Metadata flows are flows that represent any form of data flowing in the program, for example data read from a database, file etc. These do not represent a concrete value, but rather its meta representation, e.g. a column in a database table. The other group are value flows which track a possible runtime value of a variable. Even though we mentioned that we do not execute the code, we need to track string values in code, because they can be used for identification of resources, such as file or column names. In our example, these values contain names of the files that the data is read and written to.

To find the data flows, the algorithm analyzes executable invocations. An invocation consists of the body of an executable (instructions) and arguments. The arguments contain the flows for function parameters that the function was invoked with and in Python also flows of global variables, which can be also accessed. If the same function is called with two different sets of arguments, they are two different invocations.

The invocations are analyzed in an order defined by a worklist (an enhanced queue) in a loop until it is empty. At first, it contains the entry point. At the beginning of each loop, the first invocation is removed from the worklist and analyzed. When an invocation is analyzed, a so-called *executable summary* is computed. This summary contains flows of expressions at each instruction in the executable. Consider this part of the example code:

```
customer_data = get_customer_data()
```

Firstly, the flow of the return value of the `get_customer_data()` call is resolved and assigned to its corresponding symbolic expression. Next, this flow is looked up and assigned to the flow of the expression `customer_data` etc. When the processing of an invocation is finished, a complete executable summary is stored in a cache.

To keep the algorithm going, we need to add new invocations into the worklist. When the analysis finds an instruction representing an invocation, it is not processed recursively. After all, the point of using a worklist is to avoid recursion which can cause stack overflow very easily. Instead, it has to be looked up. At this point, it matters whether the invoked function comes from application code or from a library. For application functions, the summary of the invocation is looked up in the cache. If it has been already computed, it is used to resolve the flow of the return value. If not, it is added to the worklist and will be computed later.

If the function comes from library code, it is not analyzed at all. The goal is to analyze application source code, not the code of the libraries. Library code is known to us in advance, so instead of analyzing it instruction after the instruction, we can directly create the returned data flow. Python scanner contains many data flow plugins which implement flow propagations for different libraries and they are the main area of improvements of the scanner. Of course, it is not possible

to cover each function in each library, so plugin development is aiming at those libraries that are most widely used. If such function is not covered by a plugin, there is a fallback *identity handler*. It propagates flows of the arguments to the return value. It is not the most accurate propagation, but it keeps the algorithm going.

Getting back to how worklist is filled, new invocations of application executables are added at its end. When there were such invocations, the current invocation is also added to the worklist so when the new invocations are computed, the current invocation could be updated with their summaries. Also, if its summary has changed from the last time it was computed, it is necessary to recompute its callers, that is, invocations that called it. They might not be anymore in the worklist, but the changed summary of the current invocation may have an effect on them so they need to be recomputed. We can easily find callers from the call graph that was computed in input processing phase.

When there are no more changes to the summaries, the algorithm has reached a stable point and the symbolic analysis is considered to be over.

Output transformation

The result of symbolic analysis contains a lot of executable summaries. That is however not the result we can call a data lineage graph. We need to transform the result of data flow analysis to a data lineage graph. During the symbolic analysis, when we come across an invocation that represents a data input or an output such as database select or file write, on top of propagating the flow in the summary we also register it. Then, to transform the result into a graph, all we need to do is to iterate over the summaries and find the registered flows.

Flows are implemented in such a way that they track their origin, so an output flow contains the source of the data that is written out. Not all output flows contain a valid input, so some filtering has to be applied during the transformation, but eventually nodes for inputs and outputs are created along with corresponding edges. It might seem that we only need to register output flows and we can find the input flows in them, but we also register input flows in case there is an unused one (an unused input can still cause an error during program execution).

The result of the transformation is called a *connector output*. It is not yet a Manta graph, but rather a boilerplate for creating it. It contains the information about nodes that should be created and which nodes should be connected with edges. The reason for it is that all programming language scanners create a very similar output so creating a Manta graph from such output can be implemented once and used by all scanners.

Generating Manta graph

Finally, the last step of the analysis is to generate a Manta graph from a connector output. This happens in Dataflow Generator component and as already mentioned, there is one common generator for all programming language scanners. All analytic work has already been done, so the generator simply reads the connector output and creates corresponding nodes in the Manta graph. Should there be any SQL queries, it uses Dataflow Query Service to resolve them. The graph that is created is considered the output of the scanner. A visualization of

the data lineage graph created from the `json_to_csv.py` example can be seen in Figure 2.7.

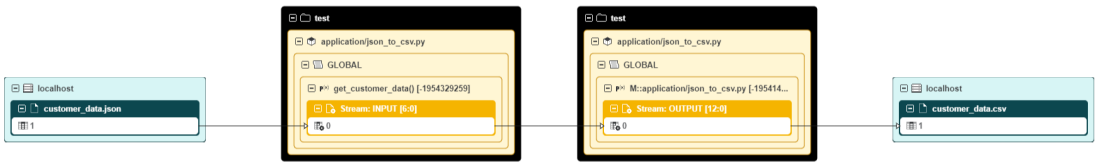

Figure 2.7: Visualization of the data lineage from the example

3. Requirements and Analysis

Following the description of Manta Flow platform, we now have enough information required to understand and discuss the problem of embedded code analysis in its context. In this chapter we will first present a motivational example for embedded code analysis, discuss requirements set by Manta Flow stakeholders, analyze multiple data technologies that support embedded code and finally we will discuss how to integrate embedded code analysis into scanners.

3.1 Purpose of embedded code analysis

We have already briefly explained what embedded code is and why it makes sense to analyze it in Manta Flow. Let us present an example that will support this explanation. It can help us build a better understanding of the motivation and illustrate some of the challenges that we need to overcome.

This example is based on a real customer inquiry. A company has decided to move a part of their systems from self-hosted solution to a cloud-based solution. The migrated system cleans, pre-processes and combines data from multiple sources to create a high-quality data source for business analyses. They decided to use AWS (Amazon Web Services) as their cloud service provider. The new pipeline uses Amazon S3, Amazon Redshift and AWS Glue. Amazon S3 is a cloud object storage which is often used to store files used by other services. Amazon Redshift is a data warehousing service based on PostgreSQL database and optimized for scalability and big data processing. AWS Glue is an ETL tool which we have already introduced and will work with later in the thesis. The transformation jobs in AWS Glue are written in Python. The new pipeline that the customer developed roughly follows these steps (shown in Figure 3.1):

1. Files containing raw unprocessed data are uploaded to Amazon S3.
2. An AWS Glue job is executed which reads these files and normalizes the data - fills empty values, normalizes column names, etc. and stores it in a common data format under curated files on Amazon S3.
3. Other AWS Glue jobs can be executed which read one or several curated files, combine the data and store it in a table in Amazon Redshift.
4. Amazon Redshift is the final destination of this data pipeline. It is then used as a data source for visualisations and analytics.

Figure 3.1: A diagram of data flows in the pipeline

Projects of this magnitude may take several months to complete, involve tens of employees who spend hundreds to thousands of man-days working on it. At this scale, it is important to keep a good overview on the progress to complete the project successfully. The customer has previously used Manta Flow to analyze different systems and would like to use it on this project as well. They would like to use it initially to watch the migration process and ensure that the new pipeline behaves correctly and no processes have been missed. After the project is over, they would like to continue using it to help with maintenance and problem resolution. The final dataset is a source for important financial and customer behavior analyses which directly impact many business decisions, so it is critical that the data is assembled correctly and data lineage visualization can help in that effort.

A careful reader might ask how data lineage analysis of embedded code can help the customer. The pipeline consists of two data storage technologies, Amazon S3 and Amazon Redshift, which are only the locations where data is stored. In order to see how values of, for example, the *average_price* column in the *Sales* table are computed, we need to look at one of the AWS Glue ETL jobs. An AWS Glue job runs a Python or Scala script on Apache Spark engine, which is a popular engine for large-scale data analytics. Therefore to provide the data lineage graph, we must run data flow analysis of the job's embedded code. When the analysis is successful, we just need to click on the *average_price* column node in the visualized graph to see the data lineage of how its value is computed. Without it we would have to locate and inspect the job manually, which might not be easy among 10s or 100s of them.

3.2 Requirements from Manta

As this work was commissioned by and developed in cooperation with Manta, the company that develops the Manta Flow platform, they have set certain requirements that the final solution should fulfill. In this section we will first state a requirement and then we will follow-up with a discussion about its motivation and implications. We will also use a new term *source technology*. In the context of embedded code analysis, the source technology is the data technology that uses embedded code. In the previous example, the source technology is AWS Glue which uses Python embedded code.

3.2.1 Functional requirements

These requirements define what the solution needs to and needs not to do.

Embedded code analysis

Provided a code script (string/file, not important implementation detail) and configuration, the service analyzes the script and delivers lineage graph for that script. This requirement is pretty straight-forward, it describes what is to be delivered.

Firstly, we shall discuss why a service is required. *Service* is a wide term and its concrete meaning depends on the context. There is already Dataflow Query

Service present in Manta Flow which handles the analysis of embedded database queries and scripts. In its core it is a Java bean (a class that encapsulates one or more objects into a single standardized object) which uses specific parts of database scanners to process the queries. It is called a service, because it is a reusable component (a bean) that can be used by any scanner and it provides its services through a standardized interface with a specific input and output. The solution described in this work, Embedded Code Service, is in many ways similar to Dataflow Query Service so it is expected to be used similarly.

The service shall accept embedded code as one of its inputs. That is different from how scanners in Manta Flow usually work, because they collect their inputs themselves. It shall also accept configuration. What will it be used for? Scanners in Manta Flow have many configurable properties which influence their execution, users may modify them in Manta Admin UI. It must be possible to configure how embedded code is analyzed using these properties, too. Additionally, from the description of embedded code we know that it is executed in the environment provided by the data technology. This environment can differ from standard runtime environment, so any differences (e.g. environment variables, pre-included libraries) need to be passed to the service in some way - in the configuration.

The last part of this requirement describes that the service shall perform data flow analysis of the input and provide the data lineage graph on the output for further processing. Contrary to the standard scanner workflow, the Manta graph should not be uploaded to Manta Flow Server, but returned in the return value.

Graph merging

The service can merge the lineage graph with the lineage graph of parent data technology when provided a node to be merged with. This requirement extends the previous one and describes what is to be done with the output.

Each scanner produces one Manta graph for one input. When it uses a service to process a part of the input that creates other graphs, they need to be merged. Furthermore, edges can be created only between nodes present in the graph, so if there is a data flow between the objects in the source technology graph and objects in the embedded code graph, they need to be merged into one so that the edge representing the data flow can be added. The merge operation is not a trivial process, so it shall be the responsibility of the service to implement it. The benefits are easy to see, no need to implement it each time the service is used across different scanners, which improves maintainability of the solution and also there is no need to understand it outside of the service. The developers may simply use the service interface to merge the results.

Multi-purpose service

The service can support multiple parent data technologies - technologies that support usage of user-defined embedded code. This requirement states that the service is multi-purpose, so there is one such service available instead of there being many services for processing embedded code in each data technology.

3.2.2 Qualitative (non-functional) requirements

On top of the definition of functionality of the solution, there are also some requirements defining its qualities.

Optimized for scanning

The service should be optimized to handle tens to hundreds of scripts for a specific combination of technology-scanner in one analysis - the limitation should be the speed of the scanner, not the speed of the service. In general, the service can be expected to be used multiple times by one scanner when it processes one connection, there may be several embedded code scripts. It is important to keep that in mind in its design. The service is intended to facilitate data lineage scanning and most of its execution time should be spent doing so. All other work that the service needs to do shall be reasonably optimized so that this overhead does not add up to a lot when it is called multiple times. This requirement does not intend to include specific scanner optimizations that can be made to improve the overall execution time of the service as they shall be evaluated after this solution is implemented and are outside of the scope of this work.

Extendibility

Extending supported data technologies should be simple, adding the support for a new data technology should only include collecting its configuration in the data technology scanner and implementing this configuration in Embedded Code Service. It is expected that the list of the scanners that use the service will grow in time based on customer requirements and available development capacity. The service shall be designed in a way that promotes extendibility so that the effort required to add the support for a new data technology is minimised. This can be done by splitting the code base to a common and a specific part so that it is obvious what needs to be added or modified in order to extend the service.

Code reuse

Maximize code reuse. Reuse the existing scanners and logic. Scanners for analyzing programming language source code are very complex. We shall use the existing scanners and make only the necessary modifications to be able to analyze embedded code. Developing one scanner for analyzing both applications and embedded code is a preferred way from resource allocation perspective.

Code duplication

Minimize code duplication, no logic should be written on more than one place. One of the purposes of the service is to hide repeating blocks of code that are tied with its usage. Deduplication improves maintainability, because when a process needs to be changed, it only needs to be changed in one place instead of in multiple location. One of the examples of such practice is graph merging logic mentioned in one of the previous requirements.

3.3 Source technology analysis

Before we dive further into the analysis, we need to examine which source data technologies, whether currently supported by a scanner in Manta Flow or planned, use embedded code. Limiting to these technologies is driven by business requirements, it does not make sense to support analysis of embedded code in a data technology that is not used by any current or prospective customer. This overview will give us a better understanding of the range of embedded code use-cases, their similarities and differences.

Hive

Hive is a distributed data warehouse system that allows users to read, write and manage big volumes of data using SQL. Starting from version 0.13.0, it supports writing user-defined functions in Java which accept parameters and return a value. The function is implemented in a class that extends a Hive `UDF` base class. Base classes define methods that shall be implemented by the function implementation and will be invoked in specific order on SQL query execution. The function class is then supposed to be packaged in a JAR that will be dynamically loaded into the Hive environment. The function is declared using `CREATE FUNCTION` SQL statement which references the JAR [1]. An example of a user-defined function implementation that turns a string into lower case can be seen in Figure 3.2.

```
1  package com.microsoft.examples;
2
3  import org.apache.hadoop.hive.ql.exec.Description;
4  import org.apache.hadoop.hive.ql.exec.UDF;
5  import org.apache.hadoop.io.*;
6
7  @Description(
8      name="ExampleUDF",
9      value="returns a lower case version of the input string."
10 )
11 public class ExampleUDF extends UDF {
12     // Accept a string input
13     public String evaluate(String input) {
14         // If the value is null, return a null
15         if(input == null)
16             return null;
17         // Lowercase the input string and return it
18         return input.toLowerCase();
19     }
20 }
```

Figure 3.2: An example of a Hive user-defined function written in Java [2]

Microsoft SQL Server

Microsoft SQL Server enables users to implement stored procedures, triggers, user-defined types, user-defined functions (scalar and table valued), and user-defined aggregate functions using any .NET Framework language, including

Microsoft Visual Basic .NET and Microsoft Visual C#. They can be implemented by arbitrary classes and methods as long as they are properly annotated. These annotations facilitate the lookup and binding between SQL Server and embedded code. Compiled code is distributed in DLL and loaded into the environment using SQL syntax [3].

SQL Server Integration Services (SSIS)

SSIS is a platform for data integration and transformation solutions. It provides graphical tools for building ETL workflows, but it is also possible to create custom objects programmatically in C# or Visual Basic. These include tasks, connection managers, log providers, enumerators and data flow components. Implementations of custom objects are expected to extend one of the base classes provided by SSIS, to override required methods and to use proper attributes. These objects are then distributed as a compiled class library. This is a similar approach as that of MSSQL [4].

```
using System;
using System.Data;
using Microsoft.SqlServer.Dts.Pipeline.Wrapper;
using Microsoft.SqlServer.Dts.Runtime.Wrapper;

[Microsoft.SqlServer.Dts.Pipeline.
    SSISScriptComponentEntryPointAttribute]
public class ScriptMain : UserComponent
{
    public override void CreateNewOutputRows()
    {
        // Accessing input column values
        string inputColumnValue = string.Empty;
        if (Variables.MyInputColumn != null && Variables.
    MyInputColumn.Length > 0)
        {
            inputColumnValue = Variables.MyInputColumn[0].ToString()
    ;
        }

        // Performing some transformation on the input column value
        string outputColumnValue = inputColumnValue.ToUpper();

        // Creating new output rows
        Output0Buffer.AddRow();
        Output0Buffer.MyOutputColumn = outputColumnValue;
    }
}
```

Figure 3.3: An example of an SSIS script component written in C#

In an example shown in Figure 3.3, a script component is defined as a class which inherits from the `UserComponent` base class provided by SSIS. The `CreateNewOutputRows` method is overridden to implement the desired logic. Within it we can access the input column values using the `Variables` object, which provides access to the input columns. In this example, the value of the first input

column (`MyInputColumn`) is retrieved and stored in the `inputColumnValue` variable. Next, a transformation is performed on the input column value (in this case, converting it to upper case), and the result is stored in the `outputColumnValue` variable. Finally, a new output row is added using the `OutputBuffer` object, and the transformed value is assigned to the output column (`MyOutputColumn`).

Snowflake

Snowflake Data Cloud is a cloud-based data storage and analytics service. It is common to use SQL to interact with data in Snowflake and embedded code is integrated in a similar way in the form of functions and stored procedures. Apart from SQL, these can be written in multiple programming languages - Java, Scala, JavaScript or Python. Each language used has (slightly) different capabilities and requirements. In case of JavaScript, it can be used to execute SQL statements and interact with the result to provide a return value. Java, Scala and Python scripts have to contain a function or a method with the first argument of type `Session` from Snowflake's Snowpark library. This argument will be populated by Snowflake when the procedure or the function is invoked and is used for interaction with Snowflake platform [5].

```
CREATE OR REPLACE PROCEDURE my_stored_procedure(param1 STRING,
    param2 INT)
RETURNS STRING
LANGUAGE PYTHON
AS
$$
    import snowpark as sp

    @sp.session
    def my_snowpark_function(session):
        # Create a DataFrame using the provided parameters
        df = session.range(0, param2).select(sp.lit(param1).alias('
    Value'))

        # Perform some transformations on the DataFrame
        transformed_df = df.withColumn('Length', sp.length(df['Value
    ']))

        # Convert the transformed DataFrame to a Pandas DataFrame
        pandas_df = transformed_df.toPandas()

        # Generate a summary string
        summary = pandas_df.describe().to_string()

        return summary

    # Call the Snowpark function with the provided parameters
    return my_snowpark_function(param1, param2)
$$;
```

Figure 3.4: An example of a Snowflake stored procedure written in Python

In an example shown in Figure 3.4, we are creating a stored procedure written in Python that takes two parameters, a string and an integer. The body of

the stored procedure contains a Python function named `my_snowpark_function`. This function uses the Snowpark session decorator to establish a session with Snowflake. Within the function, we create a Snowpark `DataFrame` and perform some transformations on it. In this example, we add a new column *Length* that calculates the length of the *Value* column. Next, we convert the transformed Snowpark `DataFrame` to a Pandas `DataFrame` and generate a summary string from it. Finally, the stored procedure calls the `my_snowpark_function` function with the provided parameters and returns the summary string.

Databricks

Databricks is a web-based data platform that combines data warehouses and data lakes with analytics build on Apache Spark and IPython-style notebooks. These notebooks are interactive computational environments. They consist of a sequential combination of cells which may contain rich text, embedded code, data visualization etc. Embedded code cells may be written in Python, Scala, SQL or R and it is possible to combine cells written in different languages in one notebook. During a notebook execution, cells written in the same language are executed in the same runtime environment and may interact with the runtime environments for other languages using the shared context of Apache Spark [6].

Let us have a notebook consisting of a Python cell (shown if Figure 3.5) and an SQL cell (shown in Figure 3.6). The Python cell sets the value of the shared variable which is then printed to the standard output. The SQL cell showcases how to access the shared variable within an SQL statement using the `$` symbol. In this example, we update the value of the `column1` column using the value stored in the shared variable.

```
# Define a shared variable
dbutils.shared.notebook.set("my_shared_variable", "Hello, Databricks
    !")

# Access the shared variable
shared_value = dbutils.shared.notebook.get("my_shared_variable")
print(shared_value)
```

Figure 3.5: Python cell of a Databricks notebook

```
-- Update a table using the shared variable in SQL
UPDATE my_table
SET column1 = $my_shared_variable
WHERE condition;
```

Figure 3.6: SQL cell of a Databricks notebook

AWS Glue

AWS Glue is a fully managed ETL service provided by Amazon Web Services (AWS). It simplifies the process of preparing and loading data for analytics, data

warehousing, and other data-related tasks. With AWS Glue, we can create and schedule ETL jobs to automate the data transformation and loading process. It handles the execution, monitoring and orchestration of these jobs. ETL jobs leverage the underlying Apache Spark framework for distributed data processing. Each job is defined in a script written in Python or Scala. This script can be written manually or the job's workflow is built in an interactive GUI and the code is generated automatically. Compared to other data technologies, AWS Glue is built entirely on embedded code. That means that each ETL job is executed entirely by a single Python or Scala script as opposed to only parts of data operations in other data technologies [7].

The example in Figure 3.7 shows the code of an ETL job that performs data processing and transformation, combining data from different sources and storing the result in an Amazon S3 bucket. Firstly, we read data from the specified database and tables into dynamic frames on which we perform data transformations. The `products` frame is modified by dropping certain fields and renaming one. Multiple joins are applied to the frames, first between `products` and `purchases`, and then between the resulting frame and `suppliers`. The resulting frame is written to an S3 location in the parquet format.

```
import sys
from awsglue.transforms import Join
from pyspark.context import SparkContext
from awsglue.context import GlueContext

glueContext = GlueContext(SparkContext.getOrCreate())

db_name = "main_db"
output_dir = "s3://glue-example/output/supply_chain"

# Create dynamic frames
products = glueContext.create_dynamic_frame.from_catalog(database=
    db_name, table_name="products_json")
purchases = glueContext.create_dynamic_frame.from_catalog(database=
    db_name, table_name="purchases_json")
suppliers = glueContext.create_dynamic_frame.from_catalog(database=
    db_name, table_name="suppliers_json")

# Keep the fields we need and rename some
products = products.drop_fields(['color', 'identification']).
    rename_field('name', 'product_name')

# Join the frames
result = Join.apply(Join.apply(products, purchases, 'id_product', '
    product_id'), suppliers, 'supplier_id', 'id_supplier')

# Write out the frame into parquet file
glueContext.write_dynamic_frame.from_options(frame = result,
    connection_type = "s3", connection_options = {"path": output_dir
    }, format = "parquet")
```

Figure 3.7: An example of an embedded script in AWS Glue

PostgreSQL

By default, PostgreSQL supports functions written in C, but theoretically users may use any language, as long as it can be made compatible with C, e.g. C++. However, that is often difficult due to different calling conventions, so its safe to assume C language is used. The function definitions are supposed to use macros from `postgres.h` header file, but otherwise are common C functions. The code is compiled and dynamically loaded into the environment using SQL. There is currently no plan to support analyzing C language, so this data technology is mentioned only for completeness and to illustrate that there are also other ways [8].

Other data technologies

Besides the data technologies already described, there are others that provide embedded code integration and are supported in Manta Flow. They follow similar principles as some that were already described before, so we will not cover them in detail. However, we list them below for completeness:

- Talend supports extending the functionalities of a Talend Job using custom Java commands [9].
- Google BigQuery supports defining functions written in JavaScript [10].
- StreamSets allows creating custom StreamSets processors in Java [11].
- Informatica supports creating custom components with Java [12].
- Azure Data Factory supports creating Custom activity with own data movement or transformation logic in C# that can be added to a pipeline [13].
- SAS supports running Python statements within a SAS session [14]. Additionally, there are multiple Python packages for interacting with SAS from Python.

3.3.1 Embedded code usage philosophy

Based on the description of embedded code usages, we can observe a few repeating patterns that will help us design Embedded Code Service.

We can see that database systems use embedded code in the form of user-defined functions or stored procedures. They can then be invoked as a part of an SQL query or an SQL script. They often return a value and may receive arguments.

Another common use-case is to define a custom transformation or a task in an ETL workflow. The details of this use-case vary more than those in database systems but in general they implement a specific interface. The methods of the interface are invoked in a pre-determined order by the data technology.

The next observation concerns the programming language being used. Java, Scala, .NET languages (mainly C# or Visual Basic) and Python are the most used programming languages. JavaScript is used occasionally and there is one

case of R and C, but we shall ignore them as there isn't currently a language scanner implemented in Manta Flow for them.

In compiled static-typed languages (Java and C#), it is common to tag the classes that shall be used as embedded code and require a rigid interface, either by extending a base class or using annotations. The code is distributed in compiled form and loaded dynamically. Interaction with the data technology is facilitated through an object that is provided as a method argument or a property of a base class.

As Python is interpreted and not compiled, it is possible to inject the embedded code into a different code to create a new script. That allows use-cases where some code is executed before embedded code which defines some variables, functions, classes etc. The embedded code may then directly read these identifiers without having to declare them, which is used to provide interface for interacting with data technology. To successfully analyze such approach, it is important to understand and simulate these assignments.

3.4 Embedded Code Service analysis

Based on the requirements set by Manta and the analysis of data technologies that use embedded code, in this chapter we are going to focus on the analysis of problems related to the implementation of Embedded Code Service in a more specific manner.

Let us first summarize what we have learned so far. Embedded Code Service is responsible for analyzing data lineage in code embedded in various data technologies and merging its lineage graph to that of the source technology. Similarly to Dataflow Query Service, it enables merging of data lineage of two cooperating technologies into one graph utilizing the existence of separate scanners for each of the respective technologies. It provides a way to compose a more complete lineage for the given technology without the need to implement the specifics of analyzing the embedded code within the source technology scanner.

We have mentioned the similarity to Dataflow Query Service so many times that we finally need to explain why it does not also handle the analysis of embedded code. They both analyze some form of embedded scripts. It would be possible to modify it for such use-case, but it would not be practical for several reasons.

Firstly, the situations in which they are used are different. Dataflow Query Service is used to analyze queries without requiring much information about them. When a query is used outside of a database, it is not always easy to detect what kind of database is being queried. Dataflow Query Service polls every scanner to decide if it can process the query and lets the first one that answers positively to do so, so this logic does not need to be implemented each time the service is used. When analyzing embedded code, this information is always known so there is no need for such process.

Secondly, the interfaces of the services do not overlap. To analyze a script, Dataflow Query Service requires connection details whereas Embedded Code Service requires a configuration object. Furthermore, interface of Dataflow Query Service contains a lot of methods used to deduce database objects. That is not needed or wanted in case of embedded code.

Due to these two reasons a decision was made to create a new service for the analysis of embedded code. Having only one would not be practical and would not bring any reasonable benefits. However, that does not mean that they have to be completely different. There are a few great solution in Dataflow Query Service that we can use so that we do not reinvent the wheel. There are quite a few more problems that we need to address before we can move on with the design and implementation. Here is the list for an overview:

1. Merging embedded code graph to the enclosing technology graph
2. Specifying runtime configuration - available libraries etc.
3. Caching

Let us now take a closer look at each of them.

3.4.1 Merging graphs together

The graphs contain a set of nodes and a set of edges between them. When two graphs are merged, the easy part is copying the nodes and edges from one graph to the other. Sometimes, embedded code uses data or data source originating in the pipeline of the source technology. We have already shown that in the SSIS example shown in Figure 3.3. In the code of the example, the function reads the input dataset and writes data to the output dataset. To create a complete lineage when merging two graphs, we also need to correctly map certain nodes of the embedded code graph to the equivalent ones in the source technology graph. This is the difficult part when two graphs are merged.

The problem is that there is not always enough information in the embedded code to resolve the input or the output. In the example, the exact structure or origin of the input dataset is not known, we only know that it contains the `MyInputColumn` column. Similarly, we only know that the output dataset has the `MyOutputColumn` column. We can easily create the column nodes from this information and connect them with an edge respecting the data flow, but the column nodes cannot exist on their own, they need to have a parent node that represents the dataset. A naive solution would be to add specific nodes representing input and output datasets of SSIS script transformation and assign them as the parents of column nodes. In the source technology scanner, we could locate such nodes in the embedded code graph and merge them together with their equivalent node (essentially, moving all edges from old node to the new node by changing edge starting or ending points). This approach would work, but is too specific. The programming language scanner needs to create specific nodes for each source technology endpoint, each source technology has to locate specific nodes and with each extension a lot of new code has to be added into both scanners. We would prefer a more generic approach.

Dataflow Query Service introduces a concept of *pin nodes* which is described in Section 2.4. This concept is well-suited to be also used in our service, although with a couple of modifications. All nodes created in a query graph are a part of the query so pin nodes serve only as external connection points which may or may not be used. In embedded code, we would like to use them as placeholders for nodes we do not know but we know they exist, so we can connect an edge to

them. These nodes could be created by a programming language scanner when it processes a data flow with unknown origin. This would alleviate the struggle to create a new type of node for a new source technology endpoint, because a generic pin node will be created each time. We have to make sure we do not create two identical pin nodes for different connection points. Pin nodes can be distinguished by their name (each node in a graph must have a name) and they only exist until the embedded code graph is merged, so they only need to be unique while embedded code is analyzed which can easily be checked by the scanner.

We have solved creation of data flows between nodes with unknown origin in embedded code, but we still need to find out how we can connect source technology graph to correct pin nodes. A naive solution could encode pin node details into its name, for example `SSIS_PIN_NODE_SCRIPT_COMPONENT_INPUT_DATASET_-MyInputColumn_COLUMN`. The source technology scanner could enumerate all pin nodes that need to be mapped, decode their names and connect them with their equivalents. This is a feasible solution, but it lacks a bit of finesse, because serializing and deserializing information into strings is a tedious process.

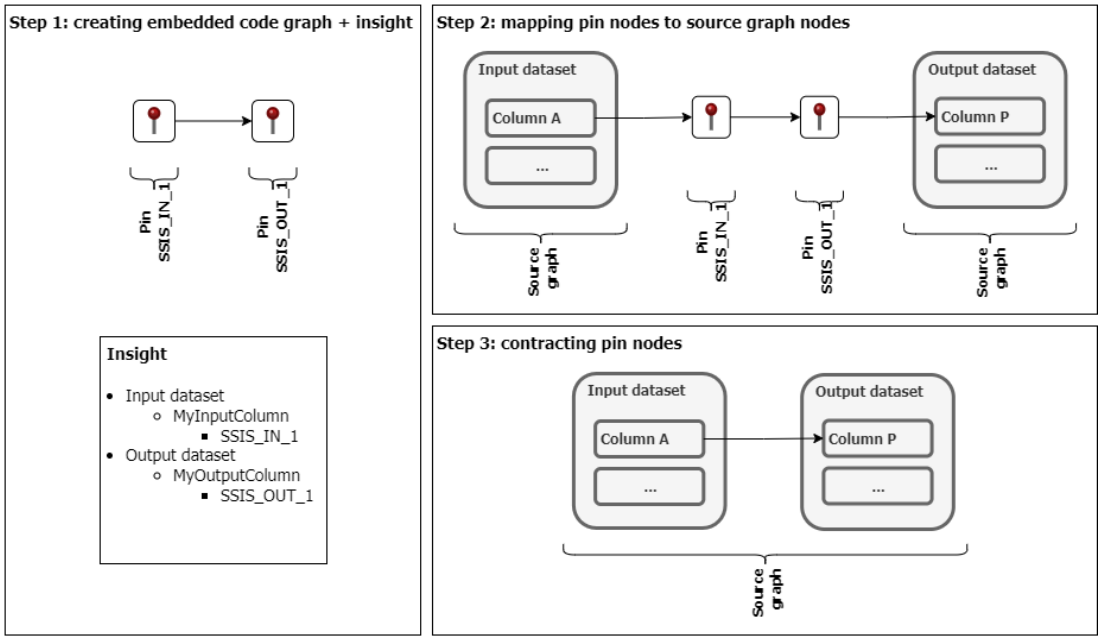

Figure 3.8: A diagram of the pin node mapping process in Embedded Code Service

What if instead we could pass a bit of information about the input to the programming language scanner and also retrieve some information about its execution after it ends? Such mechanism would help us to create a more precise graph for embedded code, because it would provide the scanner with better contextual information, but it would also allow us to pass information about the created pin nodes without having to serialize it in their names. This mechanism requires changes to be made in programming language scanners which we would like to avoid if we can, but it turns out that such mechanism can bring great benefits. The mechanism is called *Insight and Outsight* and was developed by other developers outside of the scope of this thesis. We shall describe it in more detail later when we discuss the design and implementation. For now we only need to know that the information about created pin nodes is passed to the source

technology in an *Insight* which is used to easily map them.

The mapping process is depicted in Figure 3.8. Code shown in Figure 3.3 was used for this diagram. It shows how the embedded code graph would look like when pin nodes are involved, how they are mapped using an *Insight* and the result after contracting pin nodes.

3.4.2 Specifying runtime configuration

When embedded code is executed in the source technology, the technology prepares the runtime environment in which it executes the code. This environment can often be configured by the user at which point it is necessary to propagate this information to the scanner. More importantly, some data technologies allow specifying other libraries to be included at runtime (AWS Glue, Databricks). The embedded code does not contain these libraries, they are managed by the source technology, but they can be imported and used which requires them to be included in the analysis. It means that the Extractor for the source technology needs to extract these libraries and they need to be provided to Embedded Code Service in configuration as well as the values of configurable properties.

Currently, programming language scanners are expecting that all analyzed code is present in one directory after the extraction. Because there might be additional files, which need to be analyzed, included in the configuration, the service needs to perform *input orchestration*. The goal of input orchestration is to organize the files in input directory as they would be organized in the runtime environment of the source technology. The Extractor of the programming language scanner can then use the input orchestrated in this way to produce an input for the Reader. When orchestrated successfully, the Reader would not be able to recognize any difference and would be able to analyze the code correctly.

In general, it is difficult to estimate what properties need to be passed about the runtime configuration for all technologies and languages. Furthermore, it is also not possible to perform the orchestration in a general way due to the differences of each source technology. However, we don't need to specify that in a general way. This information is only relevant for the source technology scanner, which needs to prepare the configuration, because it knows what can and needs to be provided. Embedded Code Service does not need to know the details outside input orchestration, which will be different for each data technology. Therefore, the configuration contract only needs to exist between the source technology scanner and the input orchestration component, so it can be highly customizable. Using polymorphism for the runtime configuration seems like a good choice as it can provide both a type-safe and a customizable interface.

3.4.3 Caching

It is also necessary to address the possibility of caching intermediate data. It can be expected that we will need to analyze multiple instances of embedded code in one go (multiple scripts within the source technology analysis). These instances have a lot in common. Since they are embedded, they rely on some part of the environment to be provided by the source technology and this part is often the same. It is therefore valid to consider if some analysis parts can only be computed

once and then cached and reused to improve performance.

Sadly, we were not able to find a universal answer and solution that can be implemented by Embedded Code Service. There can be some small opportunities in particular cases but we have not found a general solutions. However, these particular cases should be reviewed for each programming language scanner to see whether they can bring valuable performance benefits.

First opportunity is during input orchestration. As we mentioned earlier, part of the runtime is often the same, so when it is generated and prepared for the first time, it can be stored so it does not need to be generated again. This approach can be beneficial when the common runtime is considerably larger compared to the size of the embedded code and the duration of the analysis. For example, currently in Python scanner, parsing the standard library, which consists of approximately 2000 files, takes longer than performing the analysis of a short script (<100 LOC). In a situation when there are tens or hundreds of such scripts, the improvement could save customer a lot of time. The standard library is available for each script, therefore it could be parsed once and reused. On the other hand, for Bytecode and C# the input needs to be compiled, so each entry point has to be prepared individually.

The other reviewed opportunity is during the analysis. Since we often use the same dependencies, we could cache some intermediate results when analyzing flows in them. However, this optimization is already implemented within the scanners themselves. They use plugins for libraries that mimic the propagation of data flows in library functions and methods, so essentially, no library code is analyzed. Additionally, the execution of the worklist algorithm used for static analysis of the code is highly dependent on the inputs, so it behaves differently for each entry point and therefore there are no shared data between two entry points.

In conclusion, when implementing a new combination of technology-language in Embedded Code Service, it is advised to review the possibility of caching a part of input preparation as it can influence the overall performance of the source technology scanner. This recommendation will be a part of guidelines for implementing input orchestration for a new source technology.

4. Design And Implementation Of Embedded Code Service

Based on the analysis conducted in the previous chapter, we now have a clear understanding of what Embedded Code Service is, what is its purpose and how it is going to solve the outlined problems. In this chapter, we are going to introduce and reason about its design. We are also going to explain what changes need to be done to programming language scanners in order for them to be integrated with Embedded Code Service. At the end of the chapter we will present interesting parts of the implementation based on this design.

4.1 Embedded Code Service design

Firstly, let us summarize the steps that Embedded Code Service has to execute, because it might not be clearly obvious from previous chapters. The goal of the service is to perform data flow analysis of embedded code using one of the already existing scanners to create a data lineage graph of that code which will then be merged with the graph produced by a source technology scanner. A programming language scanner works in three steps: extraction of the input, data flow analysis and generating Manta graph. Embedded Code Service needs to perform input orchestration using the provided configuration, then launch all three stages of a scanner and after that, help to merge the graphs. The workflow looks as follows (active components are written in parentheses):

1. Input orchestration (Embedded Code Service)
2. Input extraction (scanner's Extractor)
3. Data flow analysis (scanner's Reader)
4. Generating lineage graph (Intermediate Dataflow Generator)
5. Merging graphs (source technology scanner with the help of Embedded Code Service)

4.1.1 Multiple programming languages

The first decision we have to make is whether we want to implement one universal service that can analyze embedded code written in any programming language or whether we want to have specific implementations for each programming language. This decision will greatly influence how the interface of the service is designed.

An initial idea seems to be a universal service, because that is how Dataflow Query Service is implemented. A common service promotes code reuse as multiple parts of the problem will be solved in a similar way regardless of used programming language and data technology, such as merging the graphs of embedded code and source technology. We can also find similarities between the data technologies (e.g., stored procedures written in embedded code in databases) regardless of the

programming language used. One service also means that only one component will need to be maintained. The downside of having one service is that its interface needs to be universal, so if one programming language has a different requirement or requires a specific modification, these changes will have to be reflected for other programming languages as well.

One could argue that another benefit of one service is that we have access to the analysis of any embedded code we may find, but that turns out not to be as beneficial as it may sound. In reality, the language of embedded code is always known, so it is easy to use a specific service to analyze it. Having a universal service is crucial in case of Dataflow Query Service, because it supports recognition of the SQL dialect used in the query, but in case of Embedded Code Service such feature is not needed. When implementing specific services, we can still reach similar code reuse by grouping common logic in base classes, which makes one less argument in favor of a common service. An important advantage that multiple services provide is that they not only allow the interfaces to be tailored to the needs of the programming language scanner, but they also allow different development pace for each service. The fact is that a service for Python is much more preferred by the stakeholders because of its potential and thus a lot more resources are dedicated to working on it.

Comparing the two approaches, we chose specific implementations as a more suitable solution. The last argument carries great significance due to its profound impact on the development process, so we decided to implement multiple services, each dedicated for one programming language.

Figure 4.1 shows how a specific service processes embedded code as well as components that is utilizes. The diagram is generic for any programming language.

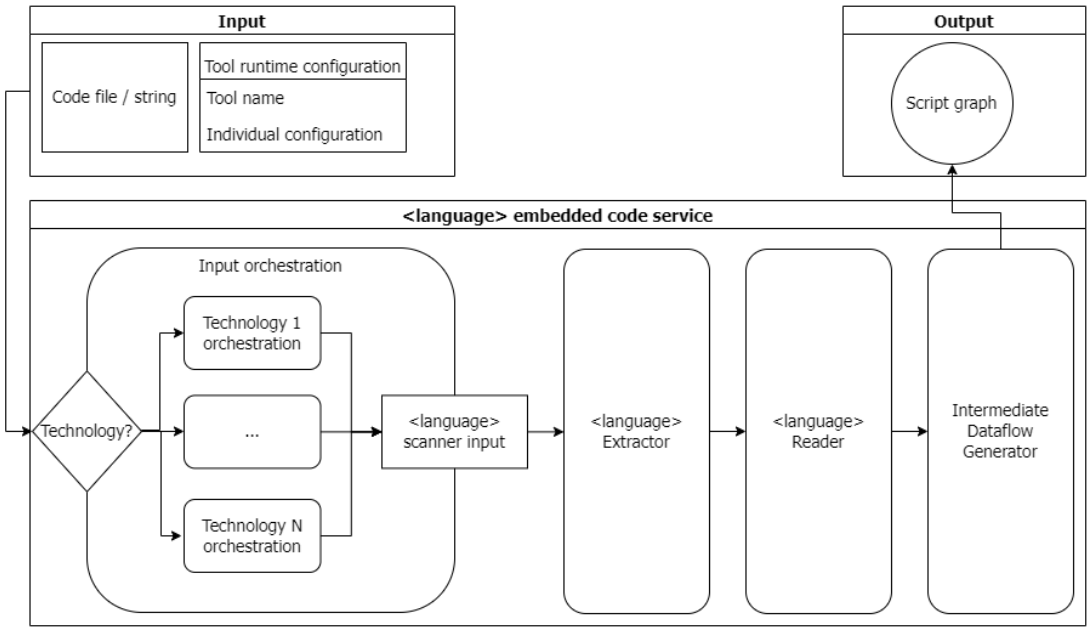

Figure 4.1: Diagram of Embedded Code Service workflow

4.1.2 Orchestration

All contemporary programming languages use some form of import mechanism where the developers can import additional libraries and frameworks. We can see that in embedded code too, often there is a mechanism for adding external libraries to be imported and used by the code. For example in AWS Glue, it is possible to define an argument of an ETL job that specifies an Amazon S3 location of a custom Python library. When AWS Glue executes a Python job, firstly it checks the S3 location and when it conforms to the required format, it copies the files to the internal working directory next to the job script. When the job is started, Python import mechanism is able to find and import these additional libraries to be used in the job script.

Additionally, some technologies (e.g. SAS, Databricks) perform additional orchestration to run the embedded code correctly, such as injecting it in a well-defined class, so this envelope does not have to be written by the user every time, adding the desired imports and only then the embedded code is run on the execution engine used by that technology. This process has to be mimicked by Embedded Code Service before the programming language scanner analyzes the code to provide an equivalent of the actual code that is executed.

The orchestration process would generally be different for each technology and programming language, but there are some common steps and some repeating patterns. This needs to be reflected in the design of orchestration so the part that needs to be implemented anew when the support for a new source technology is added is clearly distinguished from common parts. We can use *template method* design pattern that implements common parts of the process in the base class and lets extending classes redefine the specific parts. That way we clearly define what can and needs to be implemented.

4.1.3 Result

When the analysis of embedded code is finished, the service shall return the result to the source technology scanner. The result is a graph that needs to be merged with another graph and may contain pin nodes that need to be connected. Rather than returning an unfinished graph, we shall return an object that wraps it and provides an interface for connecting pin nodes and merging two graphs. The justification is simple, this graph is not always in a valid state so it should be hidden from plain view and unwanted modifications.

Pin node mapping

First thing to be done with the result is pin node mapping. This process was described in Section 3.4.1. Pin nodes can be found in the graph by enumerating its nodes and filtering those with pin node type. Each pin node can be distinguished from others by its name.

The source technology scanner performing the mapping should receive the information about created pin nodes from an Insight. A mapping simply contains a pin node and a node to be mapped to. We might want to connect pin nodes by iterating over them and for each one we create a mapping or we might want to iterate over records in an Insight, retrieve the pin node by its name and then

create the mapping. Each approach is better in different situations, so the result shall support both operations.

When a mapping is created, it is important to specify whether the pin node is an input node or an output node (relatively to embedded code). The direction decides the orientation of the edge when a pin node is merged to the source technology graph.

During the mapping process, the mapping information is stored and merging is deferred until the end so that the graph remains unchanged throughout the process.

Merging

After all pin nodes are mapped, a merge operation can start. This operation can be done only once, because it changes the source technology graph. If it was done multiple times, the graph could be damaged.

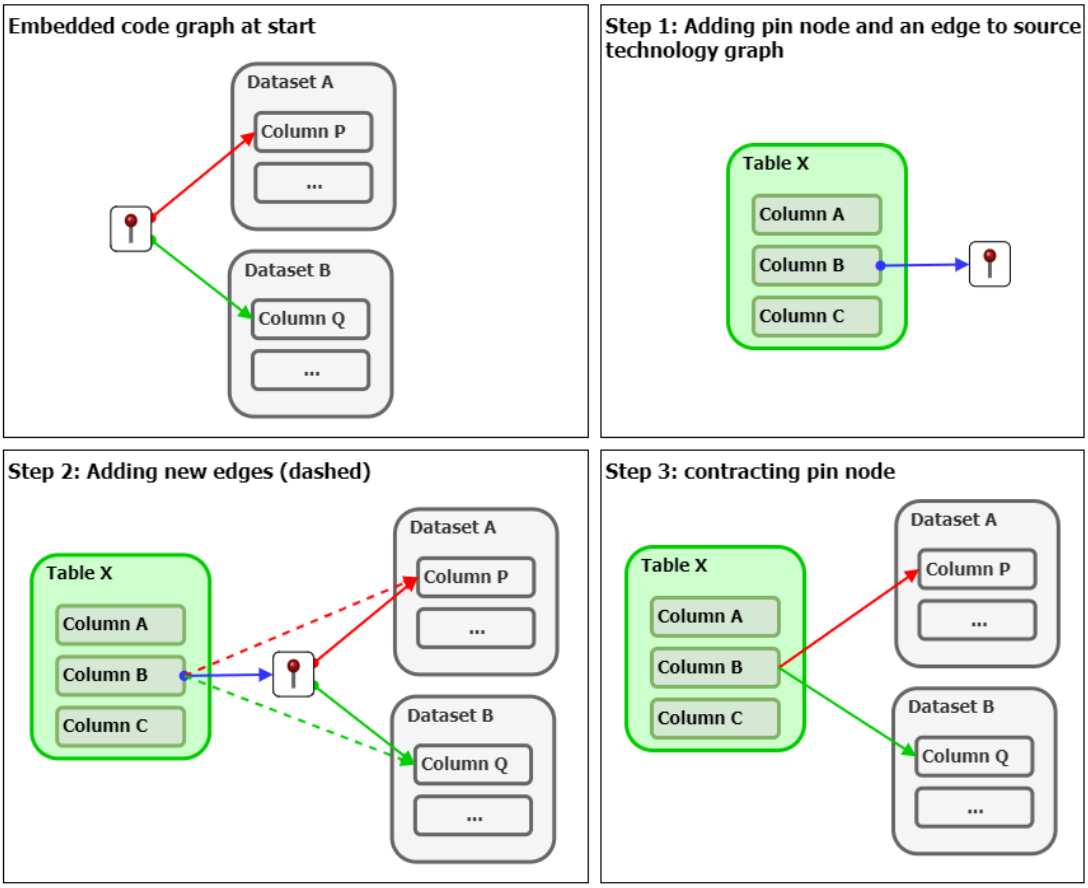

Figure 4.2: The process of merging and contracting a pin node

The merging process starts by adding the pin node into the source technology graph along with an edge that connects it to the mapped node. After that, pin node needs to be contracted and removed. That is done by first enumerating its incoming and outgoing edges and adding new edges starting at the starting points of incoming edges and ending at the end points of outgoing edges. After that, pin node and all edges associated with it are removed. A process of contracting a node can be seen in Figure 4.2.

When an edge is added to a graph, both its nodes are also added if they were not already present in the graph. After all pin nodes are added to the source technology graph and contracted, all remaining nodes and edges are added and the merging is complete.

4.2 Python scanner changes

With the design of Embedded Code Service now covered, we need to solve one more issue before we can talk about its implementation. Programming language scanners were not initially designed for running embedded code so we need to review them and design any required changes in order for them to be used in Embedded Code Service. We will be focusing on changes to Python scanner, because in this work we implemented only Embedded Code Service for Python. Other services were not marked as priorities by the stakeholders in scope of this thesis. However, since all programming language scanners follow a very similar design, the changes done on Python scanner may in the future be (with small tweaks) applied also to the remaining scanners.

4.2.1 Spring framework

Before we start with scanner changes, let us have a more detailed look on Manta Flow CLI implementation. It will help us understand some reasons for the changes. Manta Flow CLI is based on the Spring framework. This framework provides a comprehensive set of features and libraries that facilitate the development of robust and scalable Java applications. Among other essential features, we shall look at three of them that we need to understand before following further.

The first important feature is *configuration*. Spring applications typically start with a configuration phase, where developers define the application's beans, dependencies, and other settings. A bean is an object that is managed by Spring. A bean represents a reusable and configurable component of an application. It can be any Java object, ranging from plain Java objects to more complex components. Spring supports two types of configuration, XML-based configuration and Java-based configuration using annotations. Most of the components in Manta Flow CLI use XML-based configuration, however there are several new components in Manta Flow that already use the modern annotation approach.

Second key feature of the Spring framework is its support for *dependency injection*. It allows us to define the dependencies between different components (beans) of the application, and Spring takes care of injecting these dependencies at runtime. This promotes loose coupling and makes the application more modular and easier to maintain.

Lastly, the Spring framework follows the principle of *inversion of control*, which means that the framework is responsible for managing the lifecycle and execution flow of the application. In simpler terms, the application delegates control to the Spring framework, and the framework takes care of executing the appropriate components at the right time.

In Figure 4.3 we can see an example of two beans configured in XML: `userService` and `userRepository`. The `userService` bean demonstrates dependency injection by specifying the `userRepository` bean as a constructor

```
<!-- Define a Spring bean for a simple UserService -->
<bean id="userService" class="com.example.UserService">
   <!-- Inject a dependency using constructor injection -->
   <constructor-arg ref="userRepository" />

   <!-- Set a property value -->
   <property name="maxUsers" value="100" />
</bean>

<!-- Define another Spring bean for UserRepository -->
<bean id="userRepository" class="com.example.UserRepository" />
```

Figure 4.3: An example of a Spring bean XML configuration and dependency injection

dependency. When the `userService` bean is created, the Spring container will automatically inject the `userRepository` bean into it. Additionally, the `userService` bean showcases property injection by setting the `maxUsers` property to a value of `100`. The `maxUsers` property is assumed to be a property of the `UserService` class.

4.2.2 Running the scanner without scenarios

Each execution of Manta Flow CLI starts a scenario. A scenario consists of a list of tasks required to complete a certain goal. There are many different scenarios, some are used by scanners. for example various extraction and data flow scenarios, some are used for management of metadata repository in Manta Flow Server. Python scanner consists of two scenarios, *Python Extractor Scenario* and *Python Dataflow Scenario*. Extractor scenario is launched to extract input for Python scanner, dataflow scenario performs data flow analysis and generating lineage graph. Each scenario consists of generic code handling startup and completion of a scenario and specific code performing a dedicated task.

Components of Python scanner, Extractor and Reader, as well as Intermediate Dataflow Generator are currently designed specifically to be started and managed by a scenario. It is desired to modify their design and structure to enable starting the scanners from Embedded Code Service. These modifications should not be big (the scanners are already quite well designed), but they need to be finished before a language scanner can be added to the Embedded Code Service.

4.2.3 Python scanner design and Spring configuration

Python Scanner is designed to analyze user inputs. The source code is stored on the file system by users and its location together with other settings is provided in Spring configuration of the scenario when it is launched. Embedded Code Service also receives the source code, but the configuration is different for each input. As Embedded Code Service is also configured only once at the start of a scenario, this configuration has to be built at runtime, specifically in input orchestration phase.

Upon analyzing the configuration of Python scanner components we can conclude that they were designed as single-purpose components. They are constructed

with all functional elements and input configuration that they need to perform their task. Therefore, the lifecycle of such component ends when it finishes its task and a new component has to be constructed for a new task. This is feasible when Python scanner is executed from a scenario, because each scenario executes each task once. With Embedded Code Service the expected lifecycle of components is different. The service is constructed once but can be used for multiple inputs (e.g., analysis of multiple scripts that are a part of one ETL pipeline).

There are two ways how we can modify current components to support both approaches (service and CLI) efficiently. We can modify the component to become stateless. A stateless component does not hold any internal state and can be used repeatedly. Input configuration and state is managed in an instance of a task context class which is passed to the invocation of any method. The context instance can be constructed at runtime or it can be configured as a bean and injected as a dependency. This approach is best suited for components that have a single entry-point - a single method that is invoked to complete the task, but it is not a condition. In such cases the context object is passed as a method parameter during the execution.

The second approach is to create a factory (using *Factory* design pattern) for a component which will be configured in Spring to store the functional elements for the construction of the component (dependencies). The factory will allow passing the input configuration values as parameters to the factory method which constructs a new component. This approach is suitable for components which hold a complex internal state and it would be costly to refactor them to become stateless component. Also, it is a *cheap* modification, there are no changes required in the existing code, we just need to add the implementation of the factory class and this change needs to be reflected in Spring configuration.

4.2.4 Interfaces

We need to consider one more thing. The components were designed to be used by scenarios in Manta Flow CLI and are customized for that purpose. Often their interface contains a single method `execute` that performs all the steps required by the scenario. Such interface does not provide the functionality we need in Embedded Code Service, because scenarios often require additional tasks to be done that are not required by the service, e.g. serializing connector output on disk. It would be useful to decouple implementation of the functional component from the scenario interface so the functional component can be used by a scenario and Embedded Code Service but it can have its own interface to perform required tasks.

4.2.5 Components

There are three major components in Python scanner: Extractor, Reader and a shared Intermediate Dataflow Generator. These are the components that need to be modified so that they can be used both by scenarios and Embedded Code Service. Let us have a closer look at each of them and the changes that we made.

Extractor

Extractor component was already designed in a convenient way. There is a clear separation between the extractor task implementation used in the scenario (implemented in the `ExtractorTask` class) and the functional component used for the extraction (implemented in the `Extractor` class). The task uses the functional component to perform the extraction and itself implements all necessary interfaces. The design of the `Extractor` class almost satisfies our conditions, it is designed as a stateless component with the task context defined in a separate class (`ExtractorConfiguration`), the only difference is that this context is passed to `Extractor` instances in the constructor and stored. The context instance does not necessarily need to be stored in the `Extractor` instance, it was only convenient to do so in the original implementation.

The context contains several file paths pointing to the location of the input or the location of extraction folder that are read-only and need to be available only during the extraction. This component can be easily refactored to conform to the stateless component approach by removing the context from the `Extractor` class and instead providing it as a parameter to the `extract` method that performs the extraction. When used in a scenario, the context instance is already stored in the task instance that contains both context and extractor, so there are no Spring configuration changes needed apart from omitting the context as a dependency of an extractor. When used in the service, the context instance can be easily instantiated at runtime using the file paths that the service uses for input orchestration, an extractor will be injected as a dependency of the service.

Reader

Python scanner's Reader is a highly state-dependent component. It requires some effort to remove all input-specific information and collect it in a single context instance. Additionally, the current implementation requires refactoring as it does more things than it should. Reader is currently one component that implements both task interface for scenario purposes as well as data flow analysis of Python code done in the scenario.

The increased complexity of the component would make it an ideal candidate to be reworked using the factory approach, but after a deeper analysis we concluded that using this approach would only postpone important changes. The component does two tasks that are demanded by the scenario, but only the analysis needs to be done for embedded code, the other task is serialization of intermediate results for logging and debugging purposes which is not required by Embedded Code Service. Therefore also this interface needs to be fragmented at which point it is worth the time to make a complex refactoring and convert the component to a stateless one. To satisfy all required conditions we need to:

1. Decouple task interface from the Reader implementation
2. Identify stateless functional sub-components of Reader that can be reused
3. Identify state information that needs to be stored in a task context instance of Reader

We have split the original `PythonReader` class implementation into three new classes: `PythonReader`, `EntryPointAnalyzer` and `AnalyzerConfiguration`. `EntryPointAnalyzer` is a stateless functional component that performs data flow analysis of a single entry point. In Python data flow analysis, an entry point is a module or a function where data flow analysis begins. When analyzing embedded code, there is always only one well-defined entry point, in application analysis there may in general be multiple entry points. This component will be a reusable dependency of both Reader and Embedded Code Service used for analyzing individual entry points. To analyze an entry point, it requires an entry point specification and an analyzer configuration.

`AnalyzerConfiguration` is a component holding context which is used by `EntryPointAnalyzer`. It is like a small toolbox that the analyzer uses during the analysis. A new configuration has to be provided for each input, not for each entry point. The reason is that there are certain components that can be shared across multiple entry points of one input to improve performance, such as service for parsing Python code as the code files are common for all entry points, thus they only need to be parsed once. A new analyzer configuration has to be created for each input of Embedded Code Service, only one is required in a scenario.

Lastly, `PythonReader` class has been modified to work as an implementation of Reader component used by data flow scenario. Internally it uses an entry point analyzer and its own analyzer configuration to analyze all entry points one after the other defined in the input.

With these modifications, the Spring configuration had to be split in two parts so that the dependencies can be correctly resolved. First one is a general configuration that provides an `EntryPointAnalyzer` bean. The other is a scenario-specific configuration that creates the `AnalyzerConfiguration` bean and injects it to a `PythonReader` bean along with an imported analyzer bean.

Intermediate Dataflow Generator

Intermediate Dataflow Generator is a shared component that can be configured and used by any of the programming language scanners. We had to make sure that it keeps this property after all the changes. Generator has been recently refactored and the new implementation already follows many of the principles of a stateless component. However, converting it to a stateless component was not the main focus of the mentioned refactoring so we still need to do a few additional changes. There is a similar problem as in Reader that the `TransformationTask` class implements both a task for the scenario and generating data lineage graph. By splitting it into two classes, one for the purposes of the scenario and the other for generating data lineage graph, we can easily create a stateless component which can also be used by Embedded Code Service.

After the refactoring, a new `DataflowGenerator` class is the stateless component of Intermediate Dataflow Generator. `ProgramConfiguration` is a new class introduced to hold the context information about the input. Lastly, the refactored `TransformationTask` class is now just the implementation of the scenario task, previously it represented all of these new components.

While these changes of Intermediate Dataflow Generator required little code modifications, it resulted in big changes in its Spring configuration. The Spring configuration of Intermediate Dataflow Generator is layered into multiple levels

to introduce modularity. This modularity is important as there are currently 3 programming language scanners that utilize the generator and each of them is slightly different. Moreover, with the introduction of Embedded Code Service, there is another use-case with different needs. The new design reduces the amount of beans that it depends on to a bare minimum and avoids the need to define beans that are unused.

Currently, there are two possible configuration compositions. One is for scanners that are not supported in Embedded Code Service and the other for those that are. For that purpose the configuration of common beans is split in two. Base configuration sets up the stateless functional `DataflowGenerator` component that can also be used by Embedded Code Service. The other configuration sets up also context beans and scenario task bean that is required for CLI integration. These configurations are then included in scanner-specific configuration that provides scanner-specific beans.

When Embedded Code Service is not implemented for the programming language, the situation is simple as Generator is only used from a scenario, so only the scanner-specific beans need to be provided. In Figure 4.4 we can see that the configurations are *layered* one on top of another. Each *layer* creates new beans using the beans provided by the previous one, as seen in *composition* diagram. In *dependencies* diagram we can see what beans the configuration provides to the others and which it depends on. An arrow shows that a bean is available in the configuration at its beginning and is required by the configuration at its end.

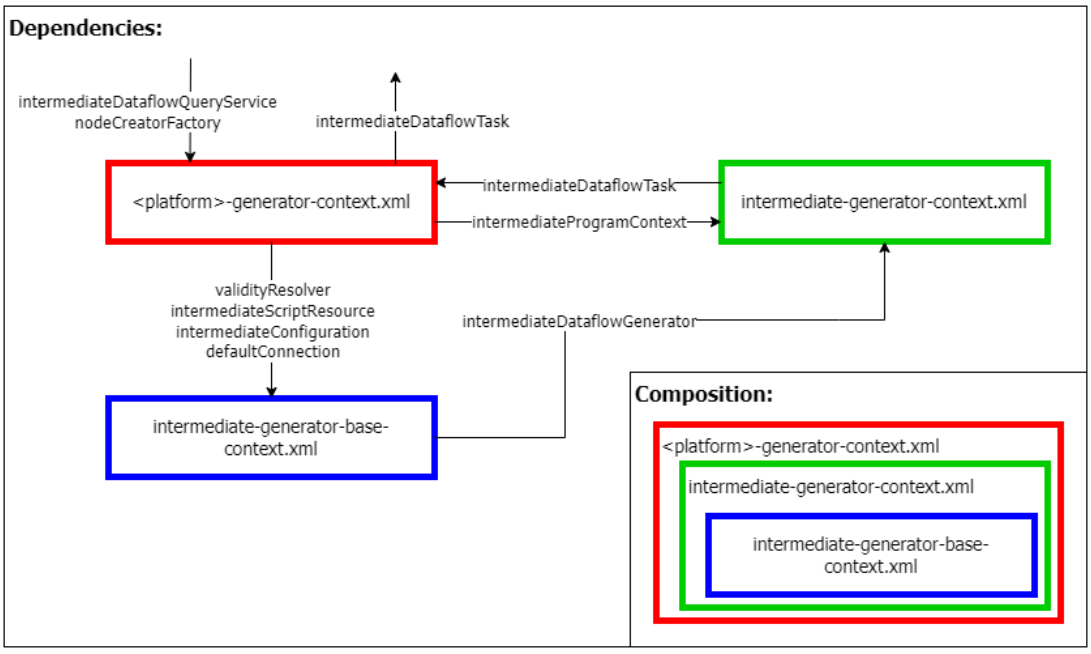

Figure 4.4: Composition and dependencies of Generator configurations when Emebedded Code Service is not supported for that programming language scanner

When Embedded Code Service is involved, there needs to be one configuration that satisfies the scenario use-case and there can be another one that satisfies Embedded Code Service needs. Embedded Code Service does not need the scenario beans, in fact, they add useless functionality for such use-case, so its best to avoid it. Moreover, in this use-case there could be multiple program configurations passed

to Intermediate Dataflow Generator during its lifecycle which cannot be defined statically in Spring. To satisfy these needs the scanner-specific configuration is split in two parts. First part is the configuration of common beans for both use-cases and then there are two specializations that utilize common beans - scenario (scanner) and service specialization.

Figure 4.5 shows a different, more complex composition of individual configurations. Similarly to the previous case, *dependencies* diagram shows what beans the configuration provides to the others and which it depends on.

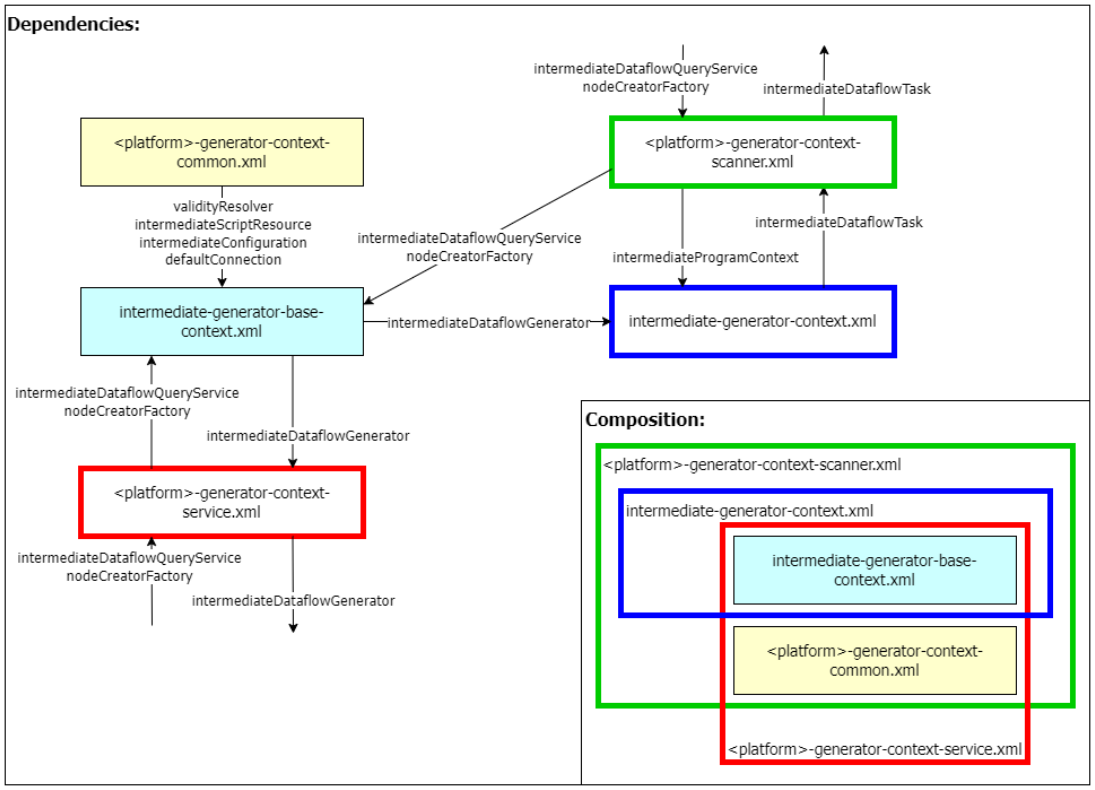

Figure 4.5: Composition and dependencies of Generator configurations when Embedded Code Service is supported for that programming language scanner

4.2.6 Insight and Outsight

There is one more change that we would like to present. Even though it was not implemented as a part of this work, it is a part of the required scanner changes to facilitate its usage in Embedded Code Service, so it makes sense to mention it. We are talking about *Insight and Outsight* mechanism that is used for pin node mapping, but can in general be used to send metadata for embedded code analysis from the source technology scanner as well as to receive additional metadata generated during such analysis to be used by the source technology scanner.

Insighter

Python Insighter is a way to provide insight to Python code analysis for external consumers. The main idea is to collect events that are important for

the external consumer, but which otherwise should not affect Python analysis. Each external consumer has to implement its own Insighter specialization together with collaborative propagation modes needed for the specific data technology. Collaborative propagation modes do not propagate data flows, but instead record events into the specialized Insighter. After the analysis is complete, an immutable *Insight* object is created which contains the recorded events and can be used by an external consumer.

We can categorize events into data events and lineage events. They need to be handled differently based on their category. Data events are events where we need to know an exact data value regardless of its origin, e.g. SQL query, database connection string etc. For the consumer, the concrete value is the important part, which is recorded in the Insight object. Lineage events are the opposite to data events. The exact value is not important in this case, but rather we track these events to create data lineage graph, e.g. reading from a file or a database. The difference from data event is that on top of recording the event into the Insighter, a pin node is created and registered as well. After the analysis ends, the external technology can utilize the recorded information to map pin nodes.

Outsight

To propagate external information for Python analysis, we can use the Insighter idea and apply it in a reversed manner. *Outsight* object represents events propagated into the analysis. It can use collaborative propagation modes to convert external events and values into Python data flows. For example, Outsight can be used to create input pins or set values of variables configured in an outside environment.

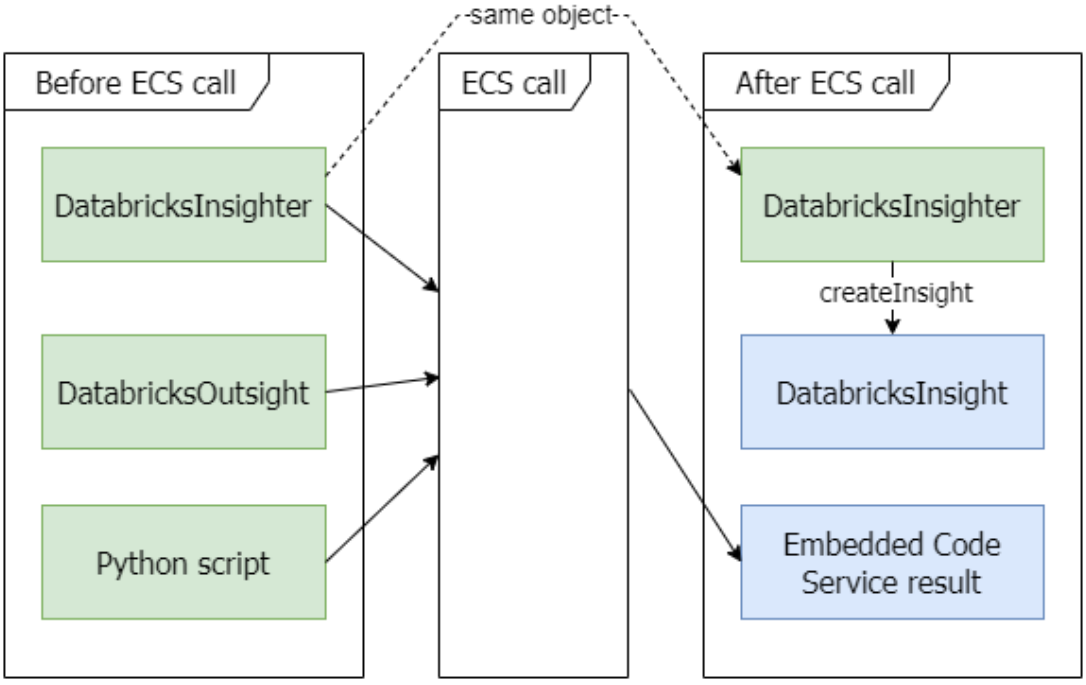

Figure 4.6: Workflow of using Insighter along with Embedded Code Service in Databricks scanner

Workflow

Let us present the workflow of this mechanism as it is used in the scanner for Databricks. This scanner was developed along with Embedded Code Service and uses it for analyzing Python code cells in Databricks notebooks.

Figure 4.6 shows how Insighter objects are created and used in Databricks scanner when calling Embedded Code Service. Before embedded code is analyzed, Databricks scanner prepares analyzed script and stores any contextual information needed for its analysis in an Outsight object. It also creates a new Insighter. These arguments are passed into the Embedded Code Service call, Insighter and Outsight are a part of configuration. After the call is complete, Embedded Code Service produces a result object. At this point, Databricks scanner can get an Insight object from the Insighter and use it to merge the result into its data lineage graph.

4.3 Python Embedded Code Service implementation

Changes implemented in Python scanner were a direct prerequisite to enable an implementation of Python Embedded Code Service. Without them the service would not be able to run embedded code analysis. Having explained these changes, we can now discuss interesting details of the implementation of this service.

Python Embedded Code Service is implemented as a standalone service intended to analyze Python embedded code. It implements two important steps - orchestrating the input in a digestible format for Python scanner and merging the resulting Python graph into the graph of the source technology. The rest of the process is delegated to Python scanner.

4.3.1 Interface

The interface of Python Embedded Code Service contains only one method: `getDataflow`. This method consumes name of the script, the content of the script, script parent node and configuration. It returns `PythonEmbeddedCodeResult` object which contains the data lineage graph of the analyzed script. The purpose of the arguments is explained in the list below:

- Name of the script is used for identifying script nodes in the graph and also for debugging purposes.
- Contents of the script can be provided in a file or in a string. A typical workflow of a source technology would prepare such scripts during its extraction and store them in files. These files can be directly used as the input or, if they are not Python source files, the code can be provided in a string.
- Parent script node is the node that represents the script in the source technology graph. All nodes in the embedded code graph will use this node as their parent for proper identification.

- The configuration stores the runtime and environment configuration for the current script. One script could be run in different environments, with different parameters or with additional user-provided libraries. Configuration that changes between scripts (is not same for each script of given technology) can be passed in this parameter. The specialized implementation of configuration only needs to implement a method that tells its kind (for safe type casting), the rest of it can be freely implemented for source technology needs.

4.3.2 Result

The return value of Python Embedded Code Service is implemented in the `PythonEmbeddedCodeResult` class. It is implemented in a similar fashion as the return value of Dataflow Query Service. It does not provide the result graph directly but instead provides an interface for connecting the two graphs. This interface contains methods for the lookup of pin nodes in the result and creating pin node mappings. After all the mappings are created, we can call the `mergeTo` method which merges result graph with source technology graph provided as an argument. After this operation, the result is considered to be consumed and cannot be used again.

4.3.3 Orchestration

The orchestration component prepares the input for data flow analysis. This is done in three steps: input orchestration, extraction and processing of the extracted input.

The component is implemented using the *template method* pattern. This pattern allows creating a template for a method where some of its steps can be reimplemented and the rest is constant. The pattern was chosen to encapsulate differences in input preparation for different source technologies into individual classes. The method works as follows:

1. Common: a temporary input directory is created. In this directory there is space for preparation of the input for the extractor as well as space for the extractor output.

2. Specific: input preparation. A specific orchestrator for data technology prepares the input based on the configuration.

3. Common: the service creates a new Extractor context and runs extraction using Python Extractor and the context. The extracted output is placed in the temporary directory.

4. Common: the service processes the extracted input by locating the entry point and creating a new analyzer configuration.

Entry point location and analyzer configuration are the outputs of orchestration and are used by entry point analyzer to analyze data flows.

4.3.4 Testing

Python Embedded Code service is covered by a set of unit and integration tests.

Unit tests verify functionality of individual classes, e.g. orchestrators storing files in correct locations or that pin node mapping is done correctly.

Integration tests verify that all components of Python Embedded Code Service are integrated and cooperate correctly. These tests run the service on different inputs and check that the data lineage graph looks as expected after the analysis and merging.

Code coverage by tests reaches 64% according to SonarQube, a code quality tool used in Manta. This number can be considered lower than standard, but we need to look why that is. The bulk of the uncovered lines are bodies of `equals` or `toString` methods along with logging messages which do not require code coverage. In combination with a low amount of code lines in Python Embedded Code Service it results in a lower coverage percentage, but the service can be considered sufficiently covered by tests.

5. AWS Glue Scanner

The previous chapters were devoted to Embedded Code Service. In this chapter, we will cover the details of the development of AWS Glue scanner prototype. AWS Glue is one of the data technologies that uses embedded code, so we will see how Embedded Code Service service can be used in data lineage analysis. To begin, we will describe AWS Glue and discuss it in terms of data lineage analysis. Next, we discuss and justify the design of a minimum viable product (MVP) of AWS Glue scanner. Finally, we will look at its implementation.

5.1 Motivation

We have already briefly introduced AWS Glue in Section 3.3. Before we dive deeper into its analysis, we shall explain why we picked AWS Glue to demonstrate Embedded Code Service capabilities.

The decision to implement Embedded Code Service in Manta Flow was motivated by the demand to analyze embedded code in several data technologies. Out of the existing programming language scanners in Manta Flow, Python scanner seemed to be the most prospective one to offer attractive value to customers for two reasons: it is widely used and the scanner yielded promising data lineage results.

At this point, there were two options for selecting the data technology for MVP implementation. We could either choose an already supported one that uses embedded Python code to extend its capabilities, or create a new scanner for an unsupported data technology. For the first option, we could choose SAS, but there was no demand for such feature as Python support has only recently been added into it. For the second option, there have long been plans for AWS Glue and Databricks scanners based on customer demand, both of which use Python extensively. Out of these two, AWS Glue uses Python in a more straight-forward way as AWS Glue jobs are in fact complete Python scripts, which was close to what Python scanner could already analyze with limited results.

5.2 AWS Glue analysis

Before we can start designing the scanner, we need to analyze how AWS Glue works, how it processes data, what metadata it stores and how we can read it etc.

5.2.1 Overview

AWS Glue is a cloud-based data integration service for discovering, cataloging and transforming data. It is a *serverless* service, which means that there is no dedicated server which requires setup or maintenance. Each execution is managed by AWS Glue which allocates computing capacity on the machines present in a data center, executes the task and then frees the resources for any other task. The customer does not need to perform any maintenance, they only pay for used computing capacity and instead may focus on developing their data processes.

AWS Glue provides its users with several key features:

1. **Data Catalog**: AWS Glue includes a centralized metadata repository, known as Data Catalog. It stores metadata information about the data sources, tables, and schemas, making it easier to discover and understand the available data.

2. **ETL jobs**: AWS Glue allows users to define and run ETL jobs using a visual interface or by writing custom code. ETL jobs enable data transformations such as filtering, aggregating, and joining data from different sources.

3. **Data crawling**: AWS Glue can automatically discover and catalog data from various sources, including databases, data lakes, and file systems. It uses crawlers to scan the data sources, infer schemas, and create tables in the Data Catalog.

4. **Data preparation**: AWS Glue provides capabilities for cleaning and preparing data before it is used for analysis. It offers built-in transformations and mappings to standardize and transform data, as well as options for creating custom transformations using Python or Scala.

5. **Integration with other AWS services**: AWS Glue integrates with other AWS services, such as Amazon S3, Amazon Redshift, and Amazon Athena, allowing users to seamlessly move and transform data between these AWS services.

These features are split into two modules: *Data Integration and ETL* and *Data Catalog*.

Data Integration and ETL

The core of data integration in AWS Glue are ETL jobs executed on Apache Spark. Apache Spark is an open-source, distributed computing system that enables fast and flexible processing and analysis of large-scale data sets. It utilizes in-memory computing to accelerate iterative computations, making it ideal for big data workloads. With its distributed architecture, Spark can seamlessly distribute data and processing tasks across a cluster of computers, enabling parallel execution and efficient resource utilization. Spark provides a rich set of libraries and APIs for various data processing tasks available in several programming languages including Python and Scala.

AWS Glue ETL jobs always contain a job script written in Python or Scala which is executed on Spark cluster configured in AWS Glue. Besides writing your own code, AWS Glue provides a tool with graphical user interface for creating ETL jobs. They can be created using transformations and data sources from the tool's toolbox. The tool auto-generates Python script based on the visualization. This script code can then be authored, but after that it can no longer be modified in the graphical tool. We can see an example of an ETL job created in this tool in Figure 5.1. This job reads a table from Data Catalog, renames some of the columns and filters out others and stores the result in a JSON file on Amazon S3, creating a Catalog table for it. Figure 5.2 shows the code generated by the tool and illustrates how ETL jobs written in Python look like.

Another part of data integration is a scheduler for running jobs periodically or on custom triggers. The jobs can also be organized in a *Workflow*, which is a different form of scheduling where multiple jobs can be executed in the order defined in the Workflow including various conditions and triggers.

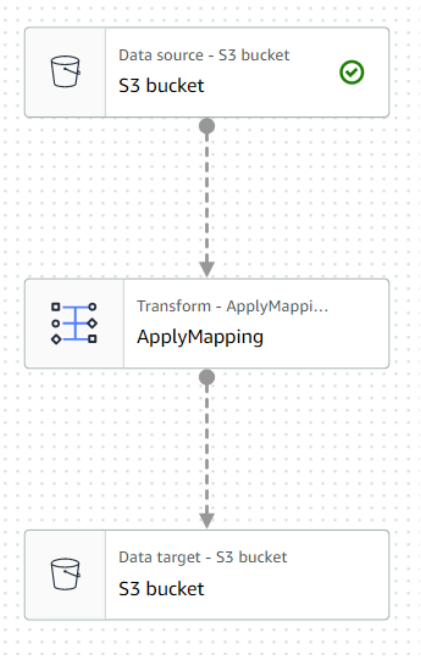

Figure 5.1: An example of an ETL job created in the graphical tool of AWS Glue

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)

# Script generated for node S3 bucket
S3bucket_node1 = glueContext.create_dynamic_frame.from_catalog(
    database="example_db", table_name="wdicountry_csv",
    transformation_ctx="S3bucket_node1"
)

# Script generated for node ApplyMapping
ApplyMapping_node2 = ApplyMapping.apply(
    frame=S3bucket_node1,
    mappings=[
        ("country code", "string", "country code", "string"),
        ("short name", "string", "short name", "string"),
        ("table name", "string", "table name", "string"),
        ("long name", "string", "full name", "string"),
```

```
        ("2-alpha code", "string", "2-alpha code", "string"),
        ("currency unit", "string", "currency", "string"),
    ],
    transformation_ctx="ApplyMapping_node2",
)

# Script generated for node S3 bucket
S3bucket_node3 = glueContext.getSink(
    path="s3://examplebucket/wdi_country_filtered/",
    connection_type="s3",
    updateBehavior="UPDATE_IN_DATABASE",
    partitionKeys=[],
    enableUpdateCatalog=True,
    transformation_ctx="S3bucket_node3",
)
S3bucket_node3.setCatalogInfo(
    catalogDatabase="example_db", catalogTableName="
    wdi_country_filtered"
)
S3bucket_node3.setFormat("json")
S3bucket_node3.writeFrame(ApplyMapping_node2)
job.commit()
```

Figure 5.2: Code script generated in AWS Glue for the ETL job created in a visual tool shown in Figure 5.1

Data Catalog

Data catalog is a centralized metadata repository than can not only be used in AWS Glue, but also in other AWS services. The purpose of Data Catalog is to organize many different data sources in a comprehensible catalog which improves data discovery and utilization. In big data environments and especially in cloud, there are many different data sources with different access rules, structure, format and schema. Data Catalog extracts this metadata from each of the data sources and stores it in an abstract structure consisting of databases, tables and columns. A user of Data Catalog then has the ability to uniformly browse and examine them regardless of the actual data source type. Data Catalog can also be used in ETL jobs to simplify reading or writing data, because each resource from Data Catalog is treated the same way in code.

New resources can be added to Data Catalog manually, but its core feature is automated data *crawling* and *classification*. Crawling is the process of exploring resources stored in a data source and classification is the process of inferring data schema in each of the resources. A common use case is to periodically crawl a bucket in Amazon S3 to discover new files and add their schema to Data Catalog using a classifier. AWS Glue provides crawlers and classifiers for most of the common data sources and formats, but it is also possible to write custom ones or use the community-provided ones from AWS marketplace.

5.2.2 Data lineage in AWS Glue

To correctly analyze data lineage in AWS Glue, we must first explore which parts of it participate in data pipelines and may contain data flows. It is easy to see that we must analyze ETL jobs as they contain ETL pipeline code, and Data

Catalog which contains metadata of data sources that could be used in ETL jobs. It turns out that this is all that we need for complete data lineage.

While Workflows seem like they could participate in data lineage, in fact they are used just for job scheduling. If jobs in a Workflow depend on each other, we can see it from just analyzing the jobs, because they would use common data sources through which they would be connected in the data lineage graph. The order in which they are executed does not play any role in data lineage analysis.

Crawlers and classifiers in Data Catalog may also contain executable code, but this code just enumerates resources present in a data source in case of crawlers and infers data schema of these resources in case of classifiers. There are no data flows in the sense of data lineage analysis, no values are moving from one place to another. They do not need to be analyzed, but we can read the results of their work in Data Catalog.

5.2.3 AWS Glue API

Now that we know which entities of AWS Glue we want to analyze, let us have a look at how we can get access to their metadata which we will need for the analysis. AWS Glue is usually managed from AWS console, which is a web application for controlling any AWS service. For programmatic access AWS services also provide rich API. It is common that any action that can be made in AWS console has an equivalent support in API. AWS is one of the biggest providers of cloud services worldwide so it should not be a surprise that there are many available SDKs (software development kits) for popular programming languages that support using APIs of AWS services. These SDKs are released under Apache License so we are free to use and distribute them and we can even modify them should we need to do so.

As Manta Flow scanners are developed in Java, naturally the first option was to explore SDK for Java. AWS Java SDK is currently in version *2.x*. This SDK contains multiple packages for each AWS service including AWS Glue. There is a comprehensible API reference and documentation for this SDK available on AWS Glue website as well as a library of examples available on GitHub which explains common usage of the SDK. Overall, this SDK is suitable to fulfill our requirements and needs. We can use it to extract all necessary metadata from AWS Glue.

Security and access permissions

It is important to understand how security and access permissions work in AWS Glue. The scanner has to read metadata of sensitive resources. It is important that it can do so in a secure way, otherwise Manta Flow users will not be willing to provide required permissions. Additionally, we need to be able to explain the users exactly what credentials and permissions we need to set up a connection for AWS Glue.

AWS provides a web service called AWS IAM (AWS Identity and Access Management) that helps managing access to AWS resources securely. IAM allows controlling and managing user identities, permissions, and access to various AWS services and resources within an organization.

With IAM, one can create and manage IAM users, groups, and roles. IAM users are specific identities that represent individuals or applications that interact with AWS services. IAM groups are collections of users with similar permissions, making it easier to manage permissions for multiple users. IAM roles are used to delegate access permissions to entities outside of organisation's AWS account, such as other AWS accounts or services.

IAM enables setting fine-grained permissions for users, allowing to control which actions they can perform and which resources they can access. It follows the principle of least privilege, which means users are granted only the permissions necessary to perform their tasks, enhancing the security of organisation's AWS infrastructure.

Overall, AWS IAM allows organisations to create tailored permissions for accessing only those AWS Glue resources that they want to analyze in Manta Flow, which builds their trust in this solution.

It is necessary to generate programmatic user credentials to be able to access AWS services from the SDK. These credentials consist of an access key ID and a secret key. This has to be done by organisation's AWS administrator.

5.2.4 ETL job metadata

Let us have a look at what ETL job metadata is available in SDK so we can determine what we can use for data flow analysis. Below is a comprehensive list of interesting properties of job metadata that are relevant for data lineage analysis. We decided to omit some non-relevant ones as the complete list is too extensive [15].

- **Name** – string that identifies a job.
- **Command** – A `JobCommand` object containing detail about the executed command.
  - **Name** – string identifying command type. There are 3 command types: `glueetl` is a standard Spark job, `pythonshell` is a job using standard Python shell (without Spark), `gluestreaming` is a streaming Spark job that runs continuously consuming data from streaming sources such as Apache Kafka.
  - **ScriptLocation** – string specifying the Amazon S3 path to a script that runs a job.
  - **PythonVersion** – string specifying the Python version being used to run a Python shell job. Python scanner only supports Python version 3.
- **DefaultArguments** – A map array of key-value pairs where each key and value are strings. Contains the default arguments for every run of this job.
- **NonOverridableArguments** – A map array of key-value pairs. Contains arguments that cannot be overriden.
- **Connections** – A list of connections used for this job.

- **GlueVersion** – string specifying AWS Glue version which determines the versions of Apache Spark and Python available in a job.

Each ETL job has a unique name that can be used for its identification. A job executes a specific command that consists of the environment, job script and arguments.

There are 3 different command types but none of them influences how Python scanner analyzes the script. Presence or absence of Spark environment can be inferred from the script. When Spark functions are used, we can expect that Spark is available, otherwise the script would fail.

Each script is stored in Amazon S3 which means that we need to download it from there in order to analyze it. Users are required to provide sufficient permissions for this action when configuring credentials.

Each ETL job can be parameterized by arguments which consist of default, non-overridable and standard arguments defined on a job run. A complete set of arguments used to run a job can only be found by examining the history of job runs. Arguments can be accessed in the script by calling a dedicated function. Some of the arguments can be used to set up the script environment for the jobs and job runs. There are 4 of them which we need to be aware of:

1. `--additional-python-modules` specifies a list representing a set of Python packages to be installed. It is possible to install packages from PyPI (Python Package Index) or provided in a custom distribution. A custom distribution entry is the Amazon S3 path to the distribution. These packages are available to be used in the job script.

2. `--extra-files` contains a list of Amazon S3 paths to additional files, such as configuration files that AWS Glue copies to the working directory of the script before running it. These files can be referenced in the script using a relative path.

3. `--extra-py-files` contains a list of Amazon S3 paths to additional Python modules that AWS Glue adds to the Python path before running the script. These modules are available to be used in the job script.

4. `--scriptLocation` contains an Amazon S3 location where the ETL script is located. This parameter overrides the script location set in job metadata.

Lastly, jobs can use certain *Connections* defined in Data Catalog. Only the Connections specified in this job parameter can be used. We will explain what they are in the following section.

5.2.5 Data Catalog metadata

We shall also look at available metadata for Data Catalog. There are three types of entities that are useful for data lineage analysis: databases, tables and connections. Firstly, let us list important properties of database metadata [16]:

- **Name** - string containing the name of the database.
- **CatalogId** - the ID of the Data Catalog in which the database resides.

The following list contains important properties of table metadata [17]:

- **Name** - string containing the table name.
- **DatabaseName** - string specifying the name of the database where the table metadata resides.
- **StorageDescriptor** - `StorageDescriptor` object describing the physical storage of table data.
  - **Columns** – An array specifying columns of the table.
  - **Location** – Location string containing URI of the physical location of the table.
- CatalogId - the ID of the Data Catalog in which the table resides

Finally, a list enumerating important properties of connection metadata:

- **Name** – string containing the name of the connection definition.
- **ConnectionType** – string specifying the connection type, one of `JDBC`, `SFTP`, `MONGODB`, `KAFKA`, `NETWORK`, `MARKETPLACE`, `CUSTOM`.
- **ConnectionProperties** - a map array of string key-value pairs. These key-value pairs define various parameters of the connection, e.g. `HOST`, `JDBC_CONNECTION_URL` etc.

Data Catalog itself is identified by Catalog ID, which is the same identifier as the 12-digit ID of the AWS account to which it belongs. AWS account could be understood as organisation's account under which all AWS resources are grouped, although it is possible for organisations to have multiple accounts. In general, an AWS account can use one AWS Glue service instance in each AWS region (geographical regions specifying data centers) and each service instance has a single Data Catalog. It is not possible to use Data Catalogs in different regions, but it is possible to use Data Catalogs belonging to different AWS accounts. A Data Catalog is therefore uniquely identified by Catalog ID (AWS account ID) and AWS region.

A Catalog contains databases which represent a logical grouping of tables. Since Data Catalog is a metadata repository, it directly does not contain any data, it just contains metadata about data sources. Data Catalog databases are just containers containing any arbitrary grouping of tables which may describe resources stored in different locations.

Tables represent a collection of related data organized in columns and rows. Each table maps to a data source for which it provides connection details, so it is possible to trace the real data location, which is crucial to provide complete data lineage. Such data source may be a relational database table, a structured file or any other data source for which there is a connector available. The benefit of a Catalog table is that no connection details have to be provided when such table is used in an ETL job or other AWS service and it also contains schema information, which is especially helpful for resources that do not directly provide it (e.g. files).

Connections can be used to store connection details for commonly used data sources such as databases. They allow secure storage of connection credentials

so they do not need to be specified in plain-text in code. This connection also becomes a single source of truth for connection details, so if its settings change, they only need to be changed in one place. A connection is specified by its type, which defines the connector that AWS Glue will use to load and save data, and a collection of connection properties. Analyzing Connections is an important metadata information for data lineage analysis, because when they are used in code, the connection details are unknown and have to be provided externally.

5.2.6 Analyzing ETL jobs

We now have enough information to think about how we can analyze data lineage in ETL jobs. The process looks rather simple. ETL jobs consist primarily of the job script, which we can analyze using Embedded Code Service. Additional metadata about the execution environment can be passed in the configuration argument of the service. This configuration shall contain argument values as well as certain Data Catalog metadata which are necessary to successfully recognize data sources used in the script. AWS Glue does not process any data outside of job scripts so at this point we will not need to map any pin nodes.

Python scanner improvements

There are certain Python scanner improvements required to support analyzing AWS Glue scripts. The scanner is already capable of analyzing Spark code as it supports the analysis of *PySpark* library (Python API for Apache Spark) function calls, but AWS Glue introduces an extension of this library called *awsglue*. This library provides additional functionality for working with Spark in AWS Glue environment and for using other features of AWS Glue such as Data Catalog. Figure 5.2 contains several function calls from this library. The scanner needs to be extended with a plugin for handling these function calls.

The main feature of this library is the `DynamicFrame` class, which is similar to PySpark's `DataFrame`. It can be created from AWS Glue Data Catalog table, connection or some other data source available in AWS and it can be converted from and to a `DataFrame`. It supports common operations over data frames such as filtering, mapping, joining, etc. There are also custom ETL transformations defined for these frames, so the users do not need to convert to `DataFrame` for common use-cases. These transformations are used when the code is generated from graphical tool in AWS Glue.

The AWS Glue `getResolvedOptions(args, options)` utility function gives access to the arguments that are passed to the script when a job is ran. Job arguments are a useful tool that makes a job dynamic and modular and are therefore used quite often. We do not have direct access to the contents of the `args` argument (that is usually supplied from `sys.argv` - command line arguments), but using Outsight we can supply it from AWS Glue scanner. Job metadata contain default arguments, it is also possible to analyze job run history and collect the different values with which the job was executed, or simply allow users to provide these values in a handy format. With this Outsight, we can simply construct a dictionary in the collaborative propagation mode containing the keys defined in the `options` argument, which is usually a list containing string constants. Figure 5.3 demonstrates the usage of this function.

```
import sys
from awsglue.utils import getResolvedOptions

args = getResolvedOptions(sys.argv, ['JOB_NAME', 'day_partition_key'
    , 'hour_partition_key', 'day_partition_value', '
    hour_partition_value'])

print "The day-partition key is: ", args['day_partition_key']
print "and the day-partition value is: ", args['day_partition_value'
    ]
```

Figure 5.3: Usage of `getResolvedOptions` function

`GlueContext` object wraps the Apache Spark `SparkContext` object, and thereby provides a mechanisms for interacting with the Apache Spark platform. It contains functions for creating data sources and data frames, working with datasets in Amazon S3, managing transactions and writing data. We are mostly interested in the functions that create and write data frames as they provide inputs and outputs of the ETL jobs. `GlueContext` can create `DynamicFrames` or `DataFrames` from Data Catalog or from options. The Data Catalog way is quite simple as only the database and table name is provided. These values are enough to match the table in the Data Catalog. Creating a frame from options is a bit trickier as there are multiple types of available connections and each of them consumes different options. However, these options are string values for which we already have a handful of mitigation strategies. Writing methods allow writing data frames in a similar way as reading them. It is possible to write both `DynamicFrames` and `DataFrames` to Data Catalog by providing name of the database and table. In advance to that, `DynamicFrame` can also be written from options or using a stored JDBC connection.

Transformation classes are a different approach to apply transformations to `DynamicFrames`. Internally, they call the relevant function on the frame. A difficult problem to solve is how the transformation is applied. Each transformation is inheriting from parent class `GlueTransform`, which declares a couple of class methods. Most of them are not interesting and only provide information to user about themselves. The interesting one is the `apply(cls, *args, **kwargs)` method, which applies the transformation with given arguments. This method is not overridden in child classes. Basically what it does internally is creating an object of the inheriting class type using the `cls` parameter. As it is a class method, this parameter is auto-filled by Python and contains the reference to the current class. Then, this object is invoked with the remaining arbitrary arguments, which in fact means that the`__call__` method of the child class is invoked. This is where the transformation method is invoked on the data frame object with the right arguments. This creates a tricky situation where each transformation uses the same `apply` method, so the current algorithm for invocation target resolution cannot reliably solve this issue. An additional improvement could be to resolve invocations based on the calling object, and if that object turns out to be a class, we can only look for the methods of that class. This solution is a mitigation strategy for an imperfect algorithm for invocation target resolution, but can be implemented rather easily.

5.2.7 Analyzing Data Catalog

It is obvious that since Data Catalog tables are used as data sources and sinks in ETL jobs, they are an integral part of the data flow. Less obvious is the fact that mapping from a table to data location (`Location` is the name of the metadata attribute containing URI to data location) also needs to be visualized. Take for example a situation where one process produces data and stores it on Amazon S3 in a file called `foo.csv` and another pipeline reads the data from Data Catalog table mapped to the S3 location of `foo.csv`, applies transformations and stores the result to a different Data Catalog table. Without the link between the data location and Data Catalog table, the graph would be disjointed.

It is also important to review whether the nodes for Data Catalog tables should exist in graph. Since the tables themselves only represent a different data source, we could replace the table node with the node of the data sources directly in the graph. While such representation would be technically correct, it would hide the semantics of using a Data Catalog table. Those users that are aware of the usage of Data Catalog tables but are unaware of the underlying data sources will not understand the graphs correctly. There may also be scenarios where only the Data Catalog tables are used in data pipelines and the underlying data source is never referenced directly. In such scenarios replacing the table node would be confusing. As we can't confirm that such situations would not occur, we cannot hide this information in the graph, therefore both nodes and the edge between them need to be created. An example of how such data lineage could be visualized is shown in Figure 5.4. This example is based on the code shown in Figure 5.2. Blue nodes represent actual files, green nodes represent Data Catalog tables for these files, yellow nodes are a part of Python data lineage.

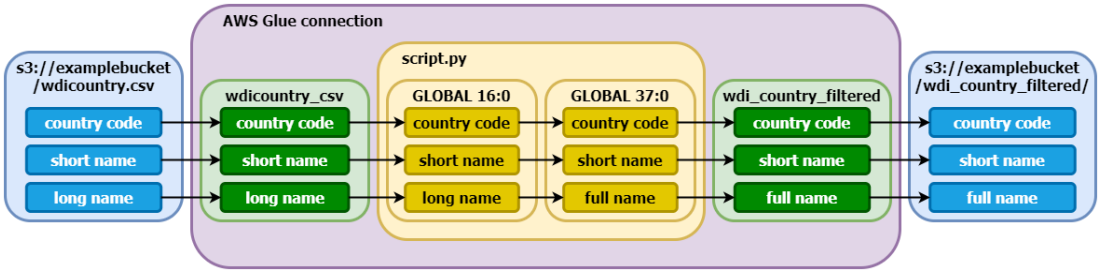

Figure 5.4: An example of data lineage graph containing Data Catalog tables

5.3 Design of AWS Glue scanner

The analysis from the previous section creates a strong foundation upon which we can build the design of AWS Glue scanner. We were thinking ahead when creating this design so it covers not only the features implemented in the MVP included in this work, but also the features that while not required for the purposes of the MVP, need to be implemented in near future to create a full-fledged scanner.

AWS Glue scanner is designed in a standard way consisting of two main components: Connector and Dataflow Generator. The task of the Connector is to connect to AWS Glue, extract all required metadata and transform it into a

general model that can be used for data flow analysis. Dataflow Generator uses this model to analyze data flows and create a data lineage graph.

5.3.1 Connection scope

Each scenario executed in Manta Flow CLI analyzes a single connection to a data technology. The first thing we need to specify is the scope of a connection in AWS Glue, that is, what entities are analyzed in one scenario execution. Usually, this scope is defined by connection URL (where available) and credentials. A scenario then analyzes everything it has access to using these connection settings. It makes sense to use this approach in AWS Glue as well. A connection in AWS Glue scanner is defined by credentials and the AWS region. This pairing uniquely identifies the AWS Glue service instance (a service instance can be identified by an AWS account ID, which we can obtain from credentials, and by an AWS region). This connection provides access to a set of ETL jobs and Data Catalog, which together represent the connection's scope of the data flow analysis. The access can be restricted by permissions set in AWS IAM service.

5.3.2 AWS Glue Connector design

AWS Glue Connector takes care of extracting and storing metadata from AWS Glue and resolving the inputs for data flow analysis. The connector is divided into 4 main components, as is common with other scanners:

1. *Extractor* which extracts metadata from AWS Glue
2. *Model* which contains the definition of the general model of the input
3. *Resolver* which is an implementation of the general model
4. *Reader* which reads extracted metadata into the model used by Dataflow Generator

Extractor

The extractor connects to the AWS Glue service instance and extracts all required metadata, saving them on the file system. To extract the metadata, AWS Java v2 SDK is used. The SDK returns data in its own Java objects. We need to extract Data Catalog metadata and ETL job metadata including job scripts. We must not forget that we also need to extract additional libraries and files which are specified in job arguments.

Extracted metadata must be stored in a convenient format and in a well-defined hierarchy on the file system so that it can be correctly loaded into the model used in the data flow analysis. It is common to store metadata in the format in which it was extracted from data technology. In a situation where retrieval of a particular artifact fails (due to insufficient rights or other error), the user can provide it manually. Users can export AWS Glue metadata in multiple formats (using `aws-cli`, a command line tool for interacting with AWS services), so we chose JSON format for convenience.

The file hierarchy for extracted metadata has to be unambiguous so that the Reader can read it correctly. We designed the file hierarchy shown in Figure 5.5. Names enclosed in `< >` represent variable names based on the actual name of the entity described between `< >` (`<job1>` would be replaced by the actual name of the first extracted ETL job etc.). The hierarchy intentionally uses `<region>` and `<account-id>` top-most directories. While a connection can only extract metadata in a single region, the name of the region is not a part of any metadata, but is required for correct naming of resources, so we store this value in the name of the directory. Account ID can be inferred from Catalog ID stored in table and database metadata, but this value is not present in job metadata and we need it to correctly resolve which Data Catalog is used in job scripts (if no Catalog ID is used when accessing Data Catalog resources, the default one is used), so we stored it in the directory name as well. However, it has another reason. It is possible to use Data Catalog belonging to another AWS account in ETL jobs, so when its metadata is extracted, it is stored in a different directory under that account's ID. Then there are two directories, `jobs` directory containing ETL job metadata and `data_catalog` directory containing Data Catalog metadata. ETL job metadata consist of metadata JSON file and script file, if the job uses additional libraries or files, they would also be stored here. Data Catalog metadata contain JSON files of databases and tables.

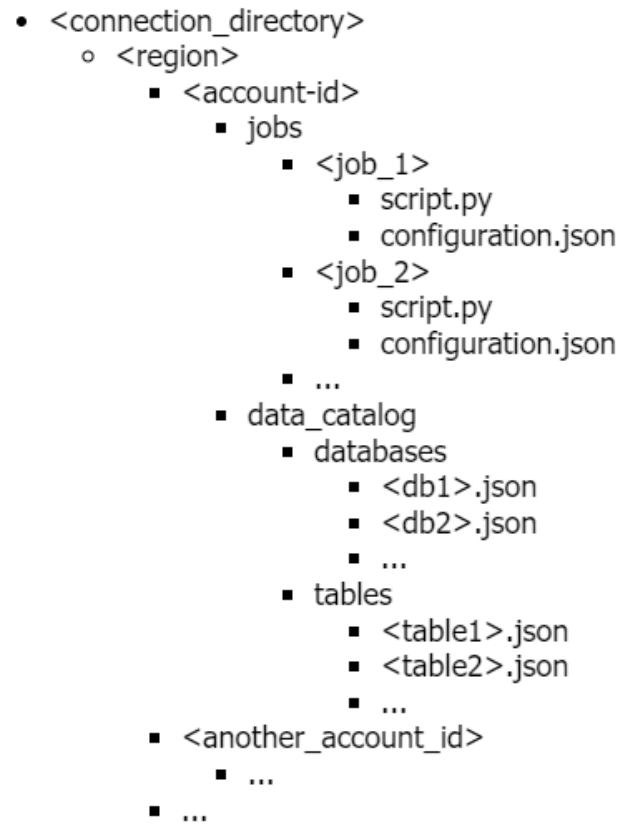

Figure 5.5: File hierarchy of extracted AWS Glue metadata

Model, Resolver and Reader

Model component defines a common data interface of the entities of the general model of AWS Glue input following the principle of loose coupling.

Resolver component contains implementations of Model interfaces. Classes are designed to be immutable so the input for the data flow analysis cannot be accidentally modified.

Reader component takes care of reading the extracted metadata and creating its object representation using the classes defined in Resolver.

5.3.3 AWS Glue Dataflow Generator design

AWS Glue Dataflow Generator analyzes data flows in the extracted inputs and creates a data lineage graph. Analyzing ETL jobs is fairly simple, all that the Generator needs to do is to call Embedded Code Service. After that it merges the result graph into the AWS Glue graph containing a node representing the job and the work is done. A more interesting problem is the analysis of Data Catalog metadata.

The goal of Data Catalog analysis is to create data flow edges between the nodes representing Data Catalog tables and the nodes representing the actual data sources as well as adding edges when these tables are used in ETL jobs. There are two possibilities how we can achieve that.

The first approach can create a more precise data lineage, but this lineage is only created when Data Catalog is used from AWS Glue (other AWS services can also use Data Catalog, for example AWS Athena can use Data Catalog tables in SQL queries). When ETL jobs are analyzed by Embedded Code Service, the resulting graph shall contain pin nodes representing reads and writes to Data Catalog tables. Then, Dataflow Generator would create the node for this table and map the pin node to the table node. Dataflow generator would also create data source node that the table is mapped to and link it with the table node. In case of Python, table schema could be passed directly to Python analysis using Outsight to provide a more detailed column-level lineage.

The second approach is a more general solution. Firstly, Data Catalog metadata would be stored in a data dictionary so it could be accessed by Dataflow Query Service. Since mapping between the data source and Data Catalog table is visualized as a data flow, it is also possible to create these flows in an extra data flow scenario. Such scenario would create data flows from data sources to Data Catalog tables for all tables present in extracted metadata. Python scanner would use Dataflow Query Service to resolve Data Catalog accesses without the need to create any extra data flows to data sources, because they would already exist. However, since there would be only dataflows from data sources to catalog tables, edges in the other direction (backlinks, when data is written to Data Catalog table) would have to be added in a postprocessing scenario.

The second solution provides more value and the lineage can also be reused for other scanners, that is why we prefer it. However, it implies that several new components need to be developed, namely:

1. *Data dictionary mapping scenario* for mapping Data Catalog metadata into a data dictionary

2. *Data Catalog data flow scenario* for creating data flows between Data Catalog tables and their data sources
3. Specific Dataflow Query Service implementation for AWS Glue Data Catalog
4. Backlink mapping configuration for adding a missing edge between Data Catalog table and data source when data is written to the table

5.4 Implementation of AWS Glue scanner

In this work we implemented the MVP of AWS Glue scanner. The goal of this MVP was not to create a full-fledged scanner, but to be able to demonstrate the functionality of Embedded Code Service. The prototype can be extended in the future following the presented design. As some of the designed features are implemented in the MVP and some are not, here is a comprehensive list that sums it up:

- Implemented features
  - Extraction of ETL job metadata and scripts
  - Data flow analysis of ETL jobs
  - Creating data lineage graph
  - Integration with Manta Flow
  - Configuration in Admin UI
  - Plugin for analyzing *awsglue* library function calls in Python scanner
  - Agent for AWS Glue
- Unimplemented features
  - Extraction of Data Catalog metadata
  - Extraction of additional files
  - Data flow analysis of Data Catalog

Let us mention interesting parts from the implementation of some of the features.

5.4.1 Extraction

AWS SDK used for metadata extraction always provides responses to requests deserialized in the form of a Java object. We have stated that we want to store metadata in a serialized JSON format. To avoid implementing serialization logic, we developed a response *interceptor* (the `GlueClientExecutionInterceptor` class) that intercepts a HTTP response before it is deserialized. At this point, the body of the response contains the metadata in the desired JSON format, so we copy it and let the response be deserialized, because it is also convenient to read some of the metadata from the provided response object.

We have also implemented a Manta Flow Agent specialization for AWS Glue extraction. Manta Flow Agent is an application for metadata extraction. In enterprises, it is common to limit access to certain networks for security reasons. Agent was created to allow extracting metadata from systems that are not accessible from the same network as Manta Flow Server.

5.4.2 Manta Flow integration

AWS Glue scanner is fully integrated with Manta Flow. It is released as one of so-called preview scanners which can be used only in preview mode. The scanner can be configured using Admin UI as all other scanners. The configuration includes specification of credentials and filtering expressions for ETL job names that should be included in the analysis.

5.4.3 Plugin for awsglue library

While not directly a part of AWS Glue scanner, we have developed a plugin for Python scanner that analyzes function calls in `awsglue` library. This plugin was important to be able to analyze Python code used in AWS Glue as the function calls are often present in it. We have conducted the analysis of this library in Section 5.2.6, which sums up the range of propagation modes that need to be implemented. We have implemented the functional base of the plugin that handles the extraction of this library and invocation of its propagation modes. We have also developed two propagation modes.

The first propagation mode handles `create_dynamic_frame_from_options` function. This function is used to read external data specified by the options into a `DynamicFrame`. The options are key-value pairs of string values and define the connection details of the data source. The propagation of this function is implemented in `CreateDynamicFrameFromOptionsPropagationMode`. The propagation mode works by trying to resolve the connection options in the best-effort way. We can split the options into two sets: stream connection options and database connection options. The propagation mode creates data read flow when it matches at least a part of the connection details and provides placeholder values for missing properties. If no option can be resolved, no flow is created. That is because we have no information about whether a stream or a database is accessed. This flow is wrapped in a flow representing an unknown column of a `DynamicFrame` and propagated to the target, which is the return value of the function.

The other propagation mode handles the `toDF` function of `DynamicFrame` class that converts this frame into PySpark `DataFrame`. Developers often prefer PySpark frames to AWS Glue frames, because they are used to them and are more capable. First, the propagation mode `DynamicFrameToDataframePropagationMode` discovers all flows representing a column in a `DynamicFrame` and transforms them to a flow representing a `DataFrame` column, preserving column name and the source of the data in the column. Transformed flows are propagated to the target, which is the return value of the function. These flows can then be used by the plugin that handles function calls of PySpark library to resolve any function calls on the returned data frame.

6. Evaluation

In this chapter, we evaluate the outcomes of the implementation presented in the preceding chapters. The primary goal of our work was data flow analysis of embedded code in the context of Manta Flow and we need to assess whether our proposed solution achieved its intended objectives.

In this evaluation, we will use an example of an AWS Glue ETL job. This example demonstrates the usage of Python Embedded Code Service for data flow analysis of embedded code in AWS Glue. It makes sense to evaluate both of these components together, because that is how they are intended to be used. We will present and explain the source of the example and then we will show and assess the resulting data lineage with respect to the predefined objectives.

Note that the presented example was created to showcase the implemented functionality and might not be a real representation of an AWS Glue job. However, it does not mean that the data flow makes no sense or that actual scripts would be completely different, but rather that they would be structured differently and contain additional logic for logging etc. The example is also limited by unimplemented features because some function calls that would be used in a production code are not yet supported in Python scanner.

6.1 ETL job example

The example that we are going to use for evaluation demonstrates a simple ETL job for data transformation using AWS Glue. The script code is shown in Figure 6.1. It showcases several features implemented in this work as well as some common features of Manta Flow to show that the implemented solution is well-integrated. At the same time, we tried to keep the example reasonable so it is similar to how ETL jobs are usually implemented.

The example is a simple ETL job defined without job arguments. It uses AWS Glue to facilitate data reading, but it does not use Data Catalog. The pipeline that we created reads data from an Amazon Redshift view into a `DynamicFrame`. This frame is converted to PySpark `DataFrame` for convenience. Next, we apply a simple data transformation to the `DataFrame`. We just added a new column, the exact transformation does not matter because its details would not be shown in the graph anyway. Python scanner does not show details about internal transformations. The transformed `DataFrame` is stored on the local file system in the CSV format. Finally, the CSV file is uploaded to Amazon S3. In more detail:

- Lines 1–4 contain library imports. In our example we need `awsglue` and `pyspark` libraries which we have already introduced and `boto3` library which is a Python library for working with AWS services.
- Line 6 defines a JDBC URL for the Redshift database that we use.
- On lines 8–18 we define the `get_df` function that uses AWS Glue context object to read data from the Redshift database. It is common to use this context to read or write data, because AWS Glue can safely facilitate this data connection using a Redshift connector. The returned data frame is

converted from AWS Glue `DynamicFrame` to PySpark `DataFrame`. This conversion is also used often as developers are more familiar with PySpark library rather than AWS Glue and it is also far more capable.

- On lines 20–21 we define data transforming function `transform_df`. As we already mentioned, the actual transformation is not important for this example, because it does not showcase any feature implemented in this work. The function is supposed to represent any set of data frame transformations.
- Lines 23–32 are the main body of the script. Firstly, we initialize Spark and AWS Glue contexts. We then retrieve input data frame using the `get_df` function and apply the transformations on it. On line 29 we store the data frame to a local CSV file using PySpark CSV writer. Because the job is executed in a serverless environment of AWS Glue, this file would be erased together with the job's working directory when the execution ends. To preserve the created file, we upload it to Amazon S3 on line 32. Although AWS Glue provides a more convenient way for working with S3 resources, many developers prefer using this common Spark approach.

```
1  from awsglue.context import GlueContext
2  from pyspark import SparkContext
3  from pyspark.sql.functions import lit
4  import boto3
5
6  jdbc = "jdbc:redshift://dev.eu-central-1.redshift.amazonaws.com
       :1234/automated_test"
7
8  def get_df(jdbcurl,table):
9      my_conn_options = {
10         "url": jdbcurl,
11         "dbtable": table,
12         "user": "masterUsername",
13         "password": "masterUserPassword",
14         "redshiftTmpDir": "s3://testdir/testbucket",
15         "aws_iam_role": "glue_execution_role"
16     }
17     df = glueContext.create_dynamic_frame.from_options(
       connection_type="redshift", connection_options=my_conn_options)
18     return df.toDF()
19
20 def transform_df(df):
21     return df.withColumn("PROFIT", lit(None))
22
23 if __name__ == "__main__":
24     sc = SparkContext.getOrCreate()
25     glueContext = GlueContext(sc)
26
27     in_df = get_df(jdbc, "sales_view")
28     csv_df = transform_df(in_df)
29     csv_df.write.csv("sales_data.csv")
30
31     s3_client = boto3.client("s3")
32     s3_client.upload_file("sales_data.csv", "mysalesbucket", "
       sales_data.csv")
```

Figure 6.1: Source code of an ETL job demonstrating embedded code analysis

The created data lineage graph can be seen in Figure 6.2. Figure 6.3 shows the same graph zoomed in on its left side, Figure 6.4 is zoomed in on the right side. Figure 6.5 shows the left part of the graph when entire Redshift database is also visualized to demonstrate that the scanner provides data lineage integrated with other data technologies. The visualized object is the AWS Glue ETL job shown in black border. The purple border represents the job script. Yellow nodes represent Python data lineage, green nodes represent files stored on a file system and red nodes represent Redshift data lineage.

We can see that the graph contains a complete data lineage as described by the example. Redshift data flows into the node representing data reading operation in Python code and into the node representing write operation. The data is stored in a file `sales_data.csv` located on localhost from where it is uploaded to Amazon S3. The data flow operation of uploading the CSV file to S3 consists of two Python nodes, because the file is first read from the file system and then uploaded to S3.

We can see that the data lineage does not contain the exact information about propagated columns. It is due to the implementation of Python scanner that currently cannot use schema information during data flow analysis. It works with a concept of unknown columns in propagations and external data source nodes are only added in Dataflow Generator.

Lastly, we would like to present performance data from data flow analysis. This data was not measured using any precise profiling technique, but rather collected by observing the logs that were produced during the execution. There are 4 interesting timestamps in the log:

1. **16:44:16.847** - AWS Glue data flow scenario started
2. **16:44:16.959** - Python scanner started analyzing embedded code
3. **16:45:02.380** - Python scanner finished analyzing embedded code
4. **16:45:02.938** - AWS Glue data flow scenario ended

We can observe that it took a little over 0.1s to initialize the AWS Glue scenario and perform input orchestration in Embedded Code Service. Python analysis took approximately 45.4s. After that, in less than 0.6s the graphs were merged, at which point the execution of Embedded Code service has finished, and the finalization steps of the scenario were executed. In conclusion, the execution of AWS Glue scanner and Embedded Code Service code took around 0.7s cummulatively (1.5% of the overall time) while the execution of Python scanner took around 45.4s (98.5% of the overall time). These results are not very precise, but we can see that Embedded Code Service is sufficiently optimised and does not create a major performance bottleneck. The effort to speed up the overall analysis should be focused mainly on Python data flow analysis.

6.2 Limitations and Future Work

As demonstrated on the example, Python Embedded Code Service designed and implemented in this work can be integrated with other data technology scanners and is capable of analyzing embedded Python code.

6.2.1 Other programming languages

We have only been able to implement the service for embedded Python code. The analysis of C# and Java code is not yet supported by any Embedded Code Service. While the implementation was not set as our goal, we have provided a comprehensive guide how these services can be developed should they be required.

6.2.2 AWS Glue scanner

We have developed a prototype of AWS Glue scanner that can analyze data lineage in basic ETL jobs. The range of developed features was sufficient to demonstrate the capabilities of Embedded Code Service, but is not yet sufficient to analyze any inputs provided by customers.

We have designed several more features that will need to be implemented in the future in order to cover basic capabilities of AWS Glue. Data Catalog analysis is a promising feature to unlock data lineage discovery in AWS Glue. The plugin for `awsglue` Python library has to be significantly extended to cover more commonly used functions and methods. The further development of the scanner will be a subject to prioritization based on customer preferences.

6.2.3 Python scanner improvements

We have implemented only a few improvements of Python scanner. We needed to redesign several components so the scanner can be integrated with Embedded Code Service and we developed an `awsglue` library plugin. These changed allowed us to analyze the example script used in this evaluation.

To provide a better value, Python scanner will need to be optimized in the future for the analysis of embedded code. We have observed that embedded Python code is often parameterized, e.g. using job arguments in AWS Glue. Python scanner currently provides poor interface for processing external values and is not optimized to analyze large sets of parameters.

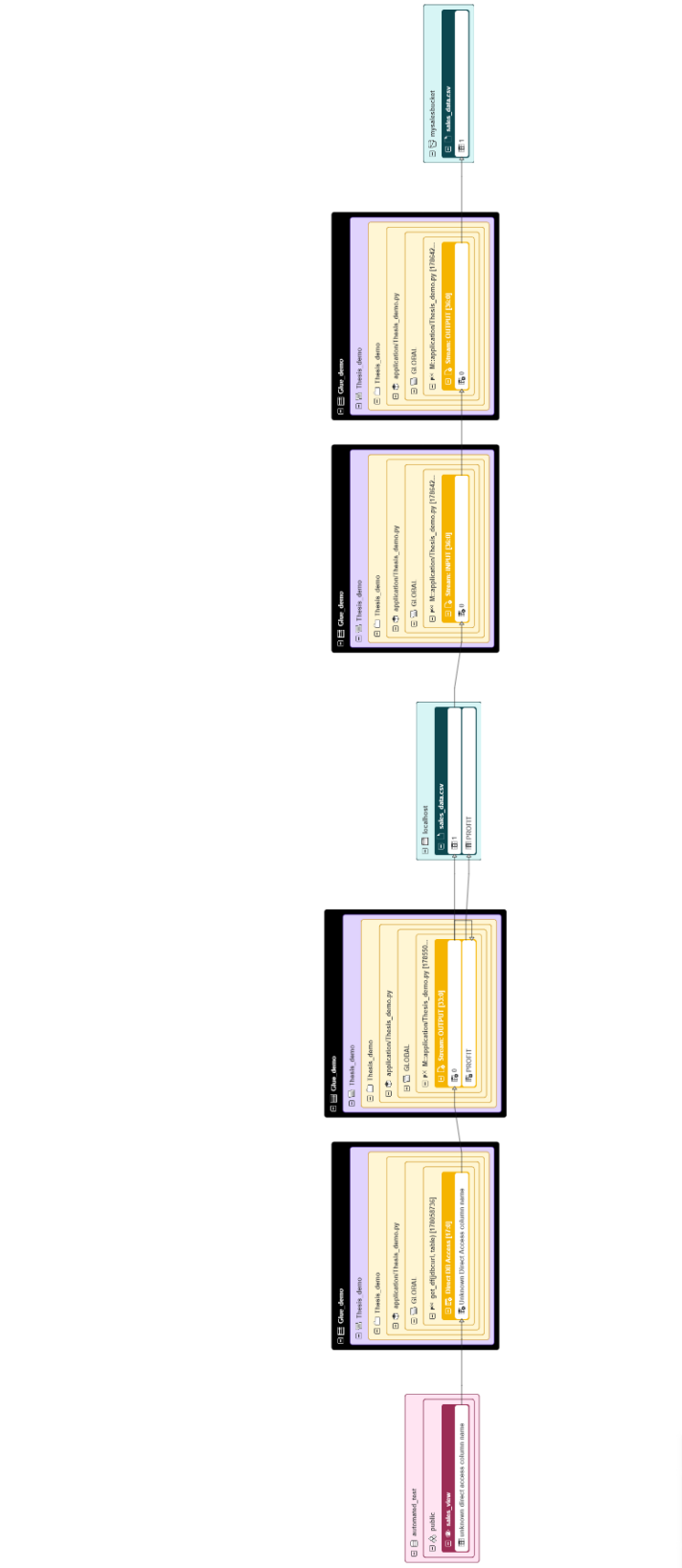

Figure 6.2: Data lineage graph created by analyzing the demonstration script

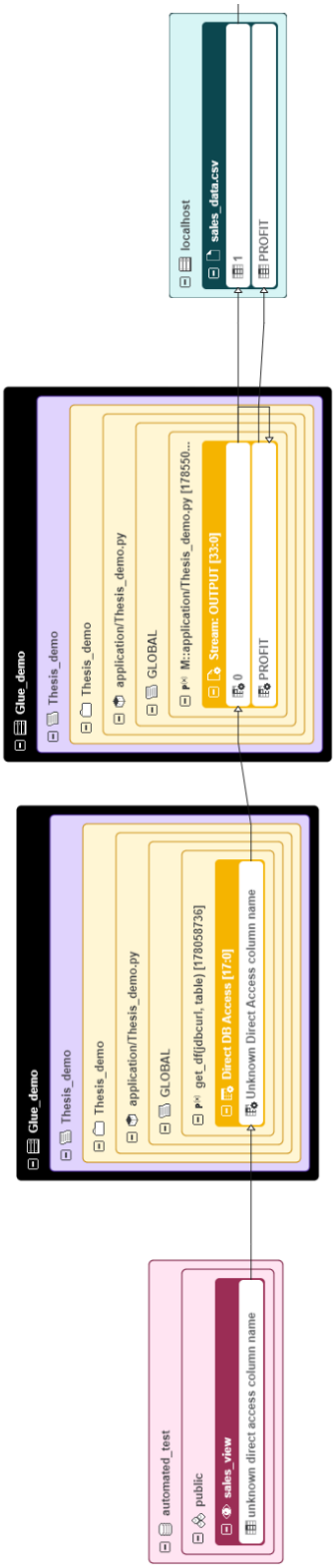

Figure 6.3: Left part of the data lineage graph

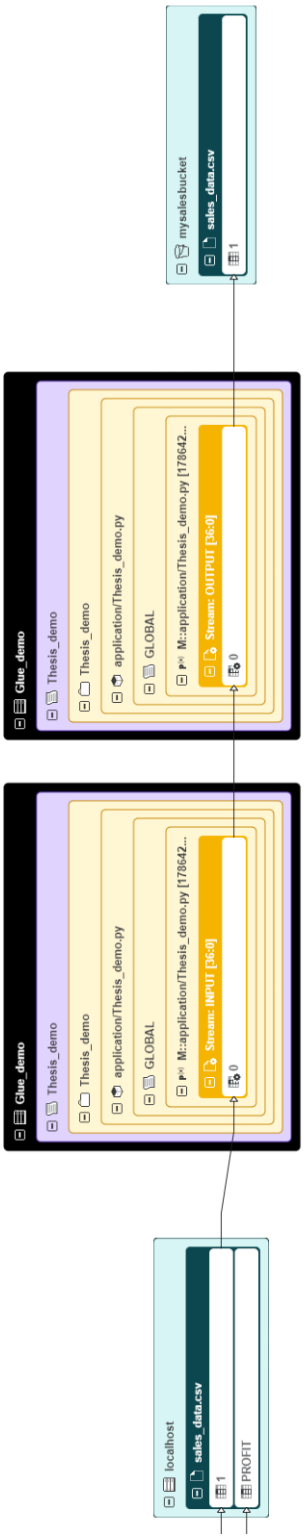

Figure 6.4: Right part of the data lineage graph

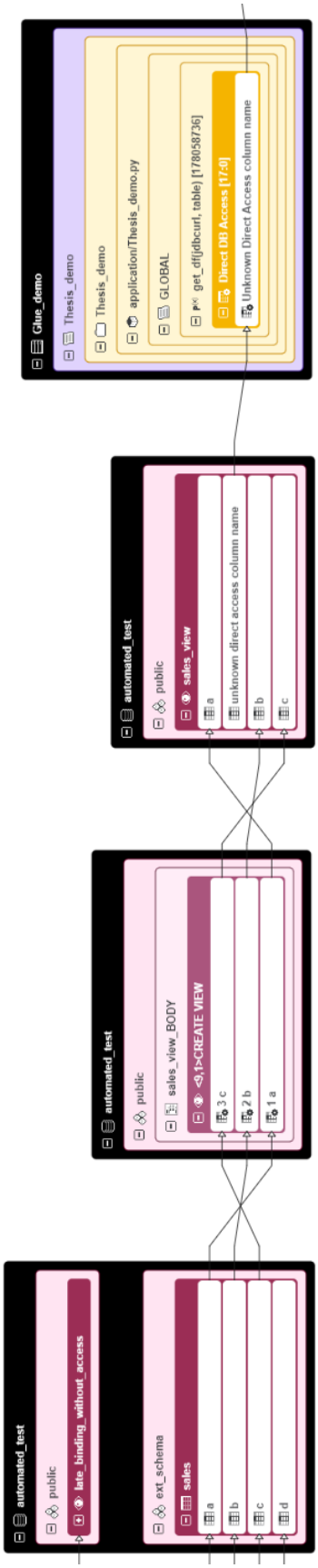

Figure 6.5: Right-most node of the data lineage graph together with a part of Redshift data lineage

7. Conclusion

In this work, we managed to successfully develop Python Embedded Code Service for data flow analysis of embedded Python code as well as a prototype of AWS Glue scanner, which uses this service to analyze ETL job scripts. The scanner is fully integrated in Manta Flow production deployment.

The service currently supports only AWS Glue, but can easily be extended to support other data technologies by implementing specific orchestration process for that technology. The analysis of embedded code is currently limited to Python code, but we have explained clear steps and necessary changes that need to be made to scanners and the service in order to support a different programming language in the future.

AWS Glue scanner prototype is able to extract and analyze data flow lineage in ETL jobs. We have conducted the analysis and designed how the scanner can be extended to also support analyzing data flows in Data Catalog.

We have extended Python scanner with a basic implementation of the plugin for analyzing function calls in `awsglue` library often used in AWS Glue ETL jobs. The plugin is ready for development of new propagation modes for function commonly used in ETL jobs.

In the last chapter we have shown that AWS Glue scanner is able to successfully analyze embedded code using Python Embedded Code Service to create a data lineage graph for AWS Glue, internally employing Python scanner for data flow analysis.

The future development should focus on finishing the unimplemented features in AWS Glue scanner to make it a full-fledged scanner and to optimize Python scanner along with Embedded Code Service to be more effective in analysis of embedded code.

Bibliography

[1] *Hive UDF*. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateFunction. (accessed: June 07, 2023).

[2] *Use a Java UDF with Apache Hive in HDInsight*. URL: https://learn.microsoft.com/en-us/azure/hdinsight/hadoop/apache-hadoop-hive-java-udf. (accessed: July 14, 2023).

[3] *CLR integration in MSSQL*. URL: https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration/common-language-runtime-integration-overview. (accessed: June 07, 2023).

[4] *Developing Custom Objects for Integration Services*. URL: https://learn.microsoft.com/en-us/sql/integration-services/extending-packages-custom-objects/developing-custom-objects-for-integration-services?view=sql-server-ver15. (accessed: June 07, 2023).

[5] *Stored Procedures Overview*. URL: https://docs.snowflake.com/en/sql-reference/stored-procedures-overview. (accessed: June 07, 2023).

[6] *Language-specific introductions to Databricks*. URL: https://docs.databricks.com/languages/index.html. (accessed: June 07, 2023).

[7] *Program AWS Glue ETL scripts in PySpark*. URL: https://docs.aws.amazon.com/glue/latest/dg/aws-glue-programming-python.html. (accessed: June 07, 2023).

[8] *C-language functions in PostgreSQL*. URL: https://www.postgresql.org/docs/current/xfunc-c.html. (accessed: June 07, 2023).

[9] *Talend custom code*. URL: https://help.talend.com/r/en-US/7.3/java-custom-code/java-custom-code. (accessed: June 07, 2023).

[10] *CREATE FUNCTION statement*. URL: https://cloud.google.com/bigquery/docs/reference/standard-sql/data-definition-language#create_function_statement. (accessed: June 07, 2023).

[11] *Creating a Custom StreamSets Processor*. URL: https://github.com/streamsets/tutorials/blob/master/tutorial-processor/readme.md. (accessed: June 07, 2023).

[12] *Developing a Custom Component in Java*. URL: https://docs.informatica.com/data-integration/b2b-data-transformation/10-4-0/engine-developer-guide/custom-script-components/developing-a-custom-component-in-java.html. (accessed: June 07, 2023).

[13] *Custom activities in Azure Data Factory*. URL: https://learn.microsoft.com/en-us/azure/data-factory/transform-data-using-custom-activity. (accessed: June 07, 2023).

[14] *Python procedure*. URL: https://documentation.sas.com/doc/en/pgmsascdc/v_017/proc/n0asd2rsj9aedgn1828aptww56of.htm. (accessed: June 07, 2023).

[15] *AWS Glue Jobs API documentation*. URL: https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-jobs-job.html. (accessed: May 30, 2022).

[16] *AWS Glue Data Catalog Database API documentation*. URL: https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-catalog-databases.html. (accessed: May 30, 2022).

[17] *AWS Glue Data Catalog Table API documentation*. URL: https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-catalog-tables.html. (accessed: May 30, 2022).

List of Figures

A. Attachments

A.1 User Documentation

To run metadata extraction and data flow analysis of AWS Glue scanner, you are required to have a computer with:

- runtime environment for Java 11 or newer installed,
- Manta Flow, version R40 or newer installed and configured.

Manta Flow requires a license to be successfully configured. Licenses are available only to Manta customers and developers, which might be a problem for the reader.

The installer for Manta Flow contains a wizard for its configuration. After the installation is successfully finished, Manta Flow can be launched.

After logging into Manta Flow Admin UI, it is necessary to create AWS Glue connection, which is listed under Data Integration tools. The connection needs to be named uniquely and requires AWS Glue credentials. The user needs to generate programmatic credentials for the AWS user that will be used to access AWS Glue. It is also possible to specify a regular expression, which filters the analyzed ETL jobs.

With connection set up, we need to create a workflow for extraction and data flow analysis. In process manager, create a new workflow and drag *Extraction* and *Analysis* steps into the workflow. They should contain your AWS Glue connection. When the workflow is created, execute it and watch the process manager to see when it completes.

After the workflow finished, you may log into Manta Flow Viewer to visualize the analyzed data lineage. In the Viewer, select the latest revision and choose, which objects you would like to visualize. Then, click *Visualize* button and explore the data lineage.

A.2 Contents of the Attachment

The list below describes the contents of the attachment:

- The *source-code* folder contains the source code developed within this work. There are only classes and other files produced in their entirety as a part of this work. Note that the source code of related Manta Flow components is not included. The source code is provided in the hierarchical structure as it appears in the development source code management tool.
- The *diff* folder contains two files in *diff* format (format used to store differences between two files) that shows the changes that were made to files not included in the *source-code* folder, but were a part of this work. The changes include the refactoring of Python scanner components for the purposes of integration with Embedded Code Service.
- The *tex-source* contains the source files used to generate PDF version of this text. It contains the original version of the diagrams and pictures shown

in figures in this work. Should any of them be difficult to read, you can examine them in better resolution.