SmartDataLake

DELIVERABLE D1.2

System Architecture

PROJECT NUMBER: 825041
START DATE OF PROJECT: 01/01/2019
DURATION: 36 months

Horizon 2020

SmartDataLake is a Research and Innovation action funded by the Horizon 2020 Framework Programme of the European Union.

Dissemination Level	Public
Due Date of Deliverable	Month 9 (30/09/2019)
Actual Submission Date	30/09/2019
Work Package	WP1 - Requirements, Architecture and Integration
Tasks	Task 1.2: Design of System Architecture
Type	Report
Lead beneficiary	ATHENA
Approval Status	Submitted for approval
Version	1.0
Number of Pages	61
Filename	SmartDataLake-D1.2-System_architecture.pdf

Abstract

This report presents the architecture of the SmartDataLake platform, specifying its individual software components and interfaces. We present the platform's architecture and its decomposition into layers, modules and components. We also detail our integration and deployment methodology, as well as the computing infrastructures to be used for this purpose.

History

Version	Date	Reason	Revised by
0.1	06/03/2019	Draft structure and ToC	Spiros Athanasiou
0.2	03/04/2019	First draft of Sections 1 – 3	Spiros Athanasiou, Dimitris Skoutas
0.3	05/06/2019	Draft input by partners	Dimitris Skoutas
0.4	02/07/2019	Revised input by partners	Dimitris Skoutas
0.5	23/08/2019	Revised input by partners	Dimitris Skoutas
0.6	03/09/2019	Draft for internal review	Dimitris Skoutas
0.7	24/09/2019	Revised draft	Dimitris Skoutas
1.0	30/09/2019	Final version	Dimitris Skoutas

Author list

Organization	Name	Contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Dimitris Skoutas	dskoutas@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Michail Alexakis	alexakis@imis.athena-innovation.gr
TU/e	Odysseas Papapetrou	o.papapetrou@tue.nl
TU/e	Hamid Shahrivari	h.shahrivari.joghan@tue.nl
TU/e	Nikolay Yakovets	n.yakovets@tue.nl
RAW	Benjamin Gaidioz	ben@raw-labs.com
UKON	Thilo Spinner	thilo.spinner@uni-konstanz.de
EPFL	Bikash Chandra	bikash.chandra@epfl.ch
EPFL	Angelos Anadiotis	angelos.anadiotis@epfl.ch

Executive Summary

This report presents the architecture of the SmartDataLake platform (*referenced as SDL from this point on*), its individual software components, and interfaces. SDL is a full software stack instantiating the emerging paradigm of a *data lake*, comprising a collection of loosely coupled components addressing the entire data exploration and analysis lifecycle in an end-to-end setting. Given the operational requirements established for SDL, and the need to accommodate existing Big Data computing infrastructures and business workflows, SDL *by design* follows an adaptive and extensible architecture allowing for the real-world deployment of the system as a whole or of selected parts of its software stack. As such, an industrial stakeholder with a data lake *already* in place will be able to harness the analytics and visualization components of SDL by integrating them in its infrastructures. In another setting, complex established data exploration and analyses workflows can exploit individual SDL algorithms by invoking the corresponding software as RESTful services.

At a high level, SDL comprises three layers, namely SDL-Virt, SDL-HIN and SDL-Vis, corresponding to the output of Work Package 2, 3 and 4 of the project, respectively. First, we find a feature-complete comprehensive data lake system, instantiated by RAW and Proteus, which can be deployed over an existing assembly of databases and file collections to provide virtualized access to the underlying data assets. This layer encapsulates and abstracts all issues related to the efficient placement, distribution, management, and retrieval of data, providing to upper layers homogeneous access to data through an SQL-like query language.

The second layer is responsible for exploring and analyzing data assets provided as *Heterogenous Information Networks* (HINs), via an assembly of *containerized* services that can be invoked independently or as parts of broader data analysis workflows. All software components in this layer are loosely coupled and are responsible for addressing any scaling requirements in a self-contained manner that suits their particular workload characteristics (elastic scaling), as well as managing any interim and output data resulting from their operation. As such, they can be invoked by providing to them as input a HIN retrieved by the first layer, a graph database, or a file, and their execution returns an output containing the analysis results and complementary files (*e.g., statistics, logs*). All components are accessible by a *common* API and programmatic bindings enabling their invocation, orchestration, and integration in diverse processing environments and workflows.

The third layer is responsible for the provision of scalable and interactive visual analytics using the data and/or analysis results provided by the two lower layers. It comprises an assembly of *containerized* services that can be invoked independently or as parts of broader exploratory data analysis and visualization workflows. The same considerations and operational settings of the second layer apply here as well – all components of this layer are loosely coupled and support elastic scaling. Execution of these components instantiates a visual interface portraying the available information to users in an appropriate interactive and intuitive manner, with any required data processing handled independently. All components are accessible by a *common* API and

programmatic bindings enabling their invocation and integration in diverse environments and workflows. Therefore, they can be embedded in existing applications or be used as standalone assets in an exploratory data setting.

With these three layers in place, our architecture comprises a number of additional components assumed to be available in an existing production setting or deployed to accommodate the needs of SDL's operation. These are not parts of the SDL platform, as they are instantiated according to operational requirements, but are considered as parts of the *Reference Architecture* of SDL, i.e., a broader collection of software artifacts and computing infrastructures available in a production setting. Indicatively, these comprise the underlying data engines and file collections, ETL pipelines, resource management system, orchestration, interactive CLI/web clients for exploratory data analysis, etc.

The Reference Architecture (*i.e., SDL platform and external components*) will be instantiated, deployed and tested in our integration and testing facilities, serving as a comprehensive deployment of SDL. This will facilitate our integration, testing, and experimental evaluation, while also providing a realistic training, evaluation and demo platform to inform industrial stakeholders about the benefits of the SDL platform. In the context of our pilots, our industrial partners will adapt this Reference Architecture according to their individual operational requirements, existing software and computing infrastructures, as well as analysis workflows, instantiating select components of the SDL platform and integrating it with any existing systems and business processes in place.

During the presentation of the SDL architecture that follows, we explore solutions to the challenges enacted by the user requirements recorded in Deliverable D1.1 "Use Cases and Requirements". Since requirements elicitation is not a static one-time process, and requirements may change through the course of the project, the design of the SDL platform may be updated accordingly to ensure it addresses its research, technical and operational requirements.

The layout of the document is as follows. In Section 1, we introduce the main concepts and challenges addressed by the project, and we outline the expected results. In Section 2, we present an overview of the SDL architecture, identify its layers, modules and components, and establish the dependencies and data flow paths between them. In Section 3, we introduce the individual software components that comprise the SDL platform and will be developed or extended during the project. For each component, we provide a short description, its role in the broader SDL platform, its main functionalities and its internal architecture. In Section 4, we present the SDL Reference Architecture and discuss the characteristics and operation of its components. Finally, in Section 5, we describe the software integration, deployment and testing practices that will be applied during software development.

Abbreviations and Acronyms

AAI	Authentication and Authorization Infrastructure
API	Application Programming Interface
CI	Continuous Integration
CLI	Command-line Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
ETL	Extract-Transform-Load
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HIN	Heterogenous Information Network
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
ML	Machine Learning
NFS	Network File System
NVM	Non-Volatile Memory
RAM	Random-access Memory
RERO	Release Early, Release Often
SDL	SmartDataLake
SLA	Service-Level Agreement
SQL	Structured Query Language
SSD	Solid-State Drive
SSH	Secure Shell
URL	Uniform Resource Locator
VM	Virtual Machine
XML	eXtensible Markup Language

Table of Contents

1. Introduction

Modern enterprises become increasingly data-driven and data-intensive, relying on data and analytics throughout the whole fabric of the business (strategic planning, sales, marketing, finance, operations) to make fact-based business decisions and to better analyse and understand business conditions. To this end, *data lakes* have emerged as *raw data ecosystems*, where large amounts of diverse structured, unstructured and semi-structured data in its natural format and in various models coexist. A data lake retains all data, including data that is kept because it might be of use at some point in the future, as opposed to predefined parts of data at predefined levels of granularity that are known in advance to serve specific purposes. Data scientists can directly tap into the data lake to analyse data from new sources, combine data of different types, come up with new business questions, test hypotheses and derive new insights and knowledge, improving flexible, fast, and ad hoc decision making.

Achieving this goal, involves several challenges: handling data heterogeneity, reducing storage costs, making sense of the data, monitoring changes, and supporting the human in the loop. The SmartDataLake project tackles these challenges by designing, developing and evaluating novel techniques, algorithms and tools for *data virtualization*, *mining of heterogeneous and evolving information networks*, and *scalable and interactive multi-faceted visualizations*.

Data virtualization enables efficient direct data accesses on heterogeneous data. Data is efficiently and adaptively accessed, queried and analysed directly in its native format, eliminating the long data-to-query times, enabling cross-format query optimizations, and accommodating data types and workloads that are not known a priori or change over time. Moreover, data can be transparently stored in different storage tiers, ranging from slow and inexpensive archives to fast but expensive main memory.

To support data exploration, an entity-centric view and organization of the data lake's contents is provided. This involves the construction and analysis of a *heterogeneous information network* that represents diverse information about various types of entities and relationships, assembled by combining information that is scattered across different data sources. Various services are provided for mining this network, including the discovery of similar or duplicate entity profiles, the ranking of entities based on multiple criteria, the prediction of new relations between entities, the discovery of communities of entities, and the monitoring of changes in the data and the analysis results.

Finally, the developed analytics tools support the *data scientist* in using her intuition and domain knowledge to steer the analysis, e.g., by feature selection and parameter tuning, while at the same time being guided by the system's own findings, e.g., the discovery of important entities, predicted links and detected communities. To this end, scalable and interactive visualizations for different types of data (spatial, temporal, graph) are provided. These include services for filtering, aggregating, ranking and summarizing information in multiple dimensions, aiming at providing different criteria and means to achieve a balance between omitting relevant information and obscuring it due to information overload. This is guided by an interactive visual analytics model

that determines the interaction paths between the data scientist and the available visualizations, allowing to concentrate on a primary facet while still hinting and linking to relevant visualizations that present different, complementary facets for the same data.

Summarizing, SmartDataLake will deliver the following concrete results:

- *Virtualized, adaptive and transparent data access and storage tiering engine*. This comprises a distributed and elastic data management system that provides functionalities for in situ query processing, adaptive indexing, data summarization, approximate query answering and transparent storage tiering.
- *Heterogeneous information network mining library*. This comprises a software library and API providing a series of algorithms for mining heterogeneous information networks. Specifically, it includes algorithms for similarity search, entity resolution and ranking, link prediction and community detection. In addition, it includes methods for detecting changes in the data and comparing analysis results between different snapshots.
- *Scalable and interactive visual analytics engine*. This comprises a software library and API for generating different types of scalable and interactive visualizations on top of the aforementioned query engine and mining library. It includes visualizations for geospatial, temporal and graph data, and it is driven by a model for interactive visual analytics determining how multi-faceted visualizations are linked together for navigation and exploration and for enabling feature and parameter space exploration and analysis.
- *Integrated SmartDataLake framework*. This comprises the complete, integrated and validated SmartDataLake framework for enabling extreme-scale analytics over sustainable data lakes. It encompasses all the novel concepts and architecture designs proposed by the project.

2. SDL Platform Architecture

In the following, we introduce the main parts of the SDL platform, presenting a high-level overview of its logical architecture and identifying the main modules and data flow paths. SDL follows a layered architecture, comprising three layers that offer *data virtualization*, *analysis of heterogeneous information networks*, and *visual analytics*, respectively.

2.1. Overview

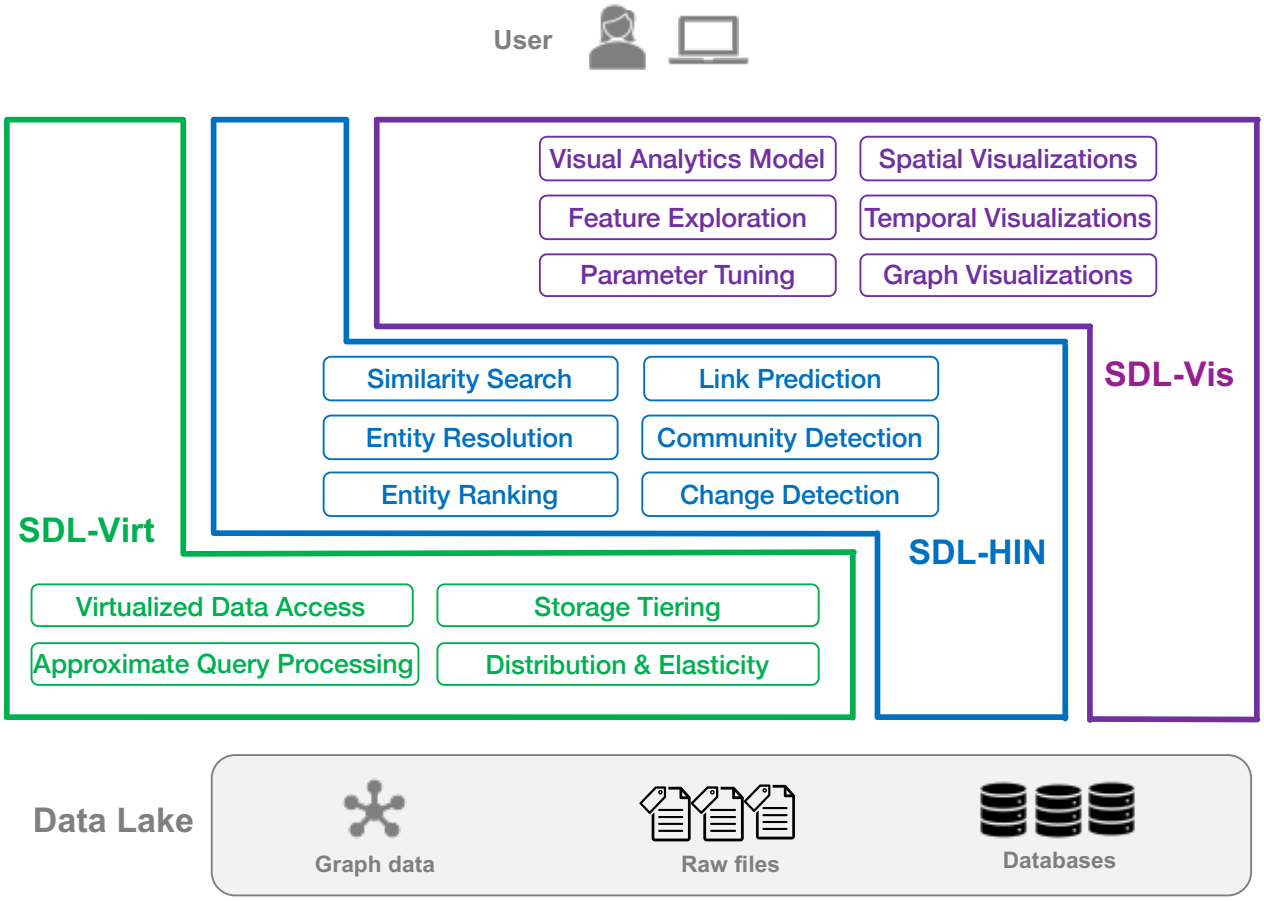

Figure 1: High-level overview of the SDL platform and its functionalities.

The SDL platform comprises the following **three main modules**, as shown in Figure 1, each one corresponding to the output of Work Packages 2, 3, and 4 of the project, respectively:

- **SDL-Virt**: A comprehensive, scalable, adaptive and highly efficient **data lake** engine providing *virtualized and elastic access* over large-scale, heterogeneous and diverse data asset collections.
- **SDL-HIN**: An extensible suite of components for the scalable *analysis and mining* of **HINs** that can be invoked independently or as part of ad hoc complex analysis workflows.
- **SDL-Vis**: An extensible suite of components providing scalable and interactive **visual analytics** of HIN assets that facilitate *exploratory data analysis* by data scientists.

2.2. SDL Modules

The modules of the SDL platform are presented in Figure 2. Each module is collapsed to its basic components, which are described briefly. A more detailed description of these components is provided in Section 3.

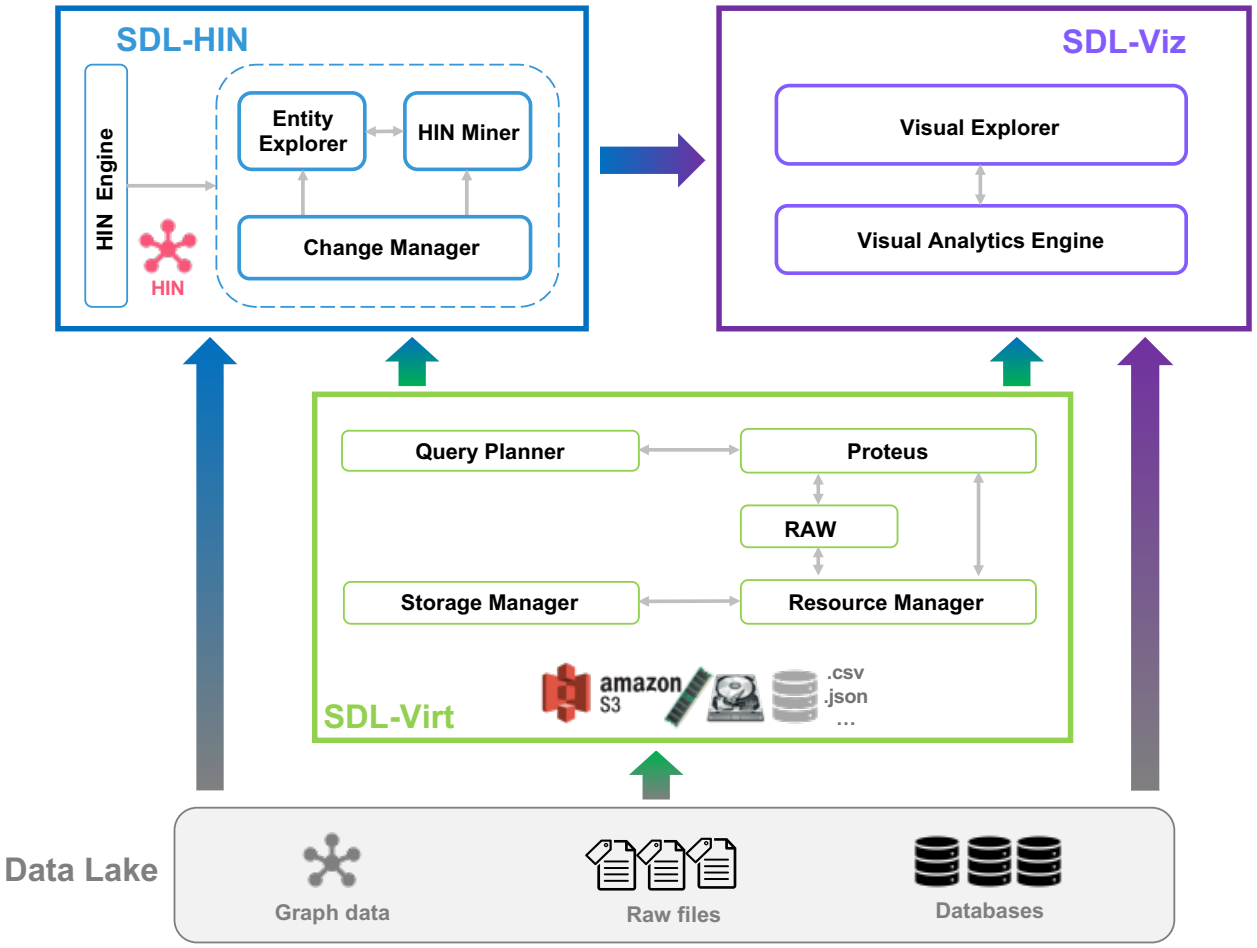

Figure 2: SDL platform modules and components.

2.2.1. SDL-Virt

SDL-Virt can be deployed over an existing assembly of databases and/or raw data files to provide virtualized access to their underlying data assets. It encapsulates and abstracts all issues related to the efficient, scalable and elastic placement, distribution, management, and retrieval of data, providing to the other layers homogeneous access to data through an SQL-like query language.

The following components constitute the SDL-Virt module (see Section 3.1 for more details on each component):

- **Query Planner**. The query planner extracts a query execution plan for the given query, taking into account the availability and location of data and resources, as well as the query characteristics (desired approximation guarantees, available access paths related to the query, etc.). The query planner also acts as a front interface of the virtualization component, providing homogenous access to the virtualized data assets available from the data lake through an SQL-like query language. The queries (typically aggregate queries) may request for either exact or approximate results. The offered API hides the diversity, heterogeneity and complexity of the underlying data, powers higher-layer components, and supports interactive and scalable computing workloads.
- **RAW**. RAW provides efficient access to raw data files. It is executed on a cluster and therefore it reads and parses files in parallel. It provides an expressive query language which enables giving strong structure to semi-structured data and applying filters, projections and aggregations.
- **Proteus**. Proteus is the execution engine that executes the query on heterogeneous data, as specified by the query plan. It is capable of executing query operators on both CPUs and GPUs as desirable and supports execution for both exact and approximate queries.
- **Resource Manager**. The resource manager allocates resources (e.g. CPUs, GPUs and memory) for the execution of query operators.
- **Storage Manager**. The storage manager handles optimal storing and replication of alternative representations of the data between different storage layers (e.g., RAM, local hard disk, network storage), in order to increase the expected query throughput.

Based on these components, the querying pipeline is as follows. The SQL-like query enters the module from the API offered by the Query Planner. The planner generates a logical execution plan and passes it to the first execution engine layer, Proteus. For relational data, Proteus executes the plan directly. The part of the plan involving raw data is passed to RAW, for more efficient execution. Both Proteus and RAW have direct access to the Resource Manager and to the Storage Manager, which transparently handles and combines local storage, network storage (e.g., S3 and HDFS), and in-memory data. After the plan is fully executed, the results are saved and then a handler is passed to the user as a response through the Query Planner's API.

2.2.2. SDL-HIN

SDL-HIN is responsible for exploring and analyzing HIN data assets via an assembly of containerized services (**HIN Analytics**) that can be invoked independently or as parts of broader data analysis workflows. All software components in this layer are loosely coupled and are responsible for addressing any scaling requirements in a self-contained manner that suits their particular workload characteristics, as well as managing any interim and output data resulting from their operation. As such, they can be invoked by providing to them as input a HIN retrieved by the first layer, a graph database, or a file, and their execution returns an output containing the analysis results and complementary files (e.g., statistics, logs). Each component is accessible by an API and programmatic bindings enabling their invocation, orchestration, and integration in diverse processing environments and workflows.

The following components constitute the SDL-HIN module (see Section 3.2 for more details on each component):

- **HIN Engine**. This component manages the HIN-induced graph and provides all other components of the SDL-HIN module with homogeneous access to HIN data assets queried through the Data API of SDL-Virt or provided directly as input (i.e., passthrough).
- **Entity Explorer**. This component operates over *collections of entity profiles* (HIN nodes), offering two high-level functionalities. The first is *similarity search*, which aims at discovering similar entity profiles within a dataset or across datasets. The second is *entity resolution*, which aims at identifying entity profiles that correspond to the same real-world entity.
- **HIN Miner**. This component operates over a *HIN*, offering functionalities for *entity ranking*, i.e., assigning importance scores to entities in the network, *link prediction*, i.e., predicting or recommending new links between entities in the network, and *community detection*, i.e., discovering groups of related or interacting entities in the network.
- **Change Manager**. This component monitors the input data to detect changes and accordingly trigger updates to the analysis results. It also tracks different network centrality measures that are useful for community detection and auxiliary structures useful for similarity search and entity resolution.

2.2.3. SDL-Vis

SDL-Vis is responsible for the provision of scalable and interactive visual analytics for HINs and/or the output of the query and analysis components of the lower layers. It comprises an assembly of containerized services (spatial/temporal/network visualizations; exploration & tuning) that can be invoked independently or as parts of broader exploratory data analysis and visualization workflows. All visualization components are loosely coupled and are responsible for addressing any scaling requirements in a self-contained manner, as well as managing any interim and output data resulting from their operation. Further, they are invoked by providing to them as input a HIN, a file, or the output of an analysis component. Their execution instantiates a visual interface portraying the available information to users in an appropriate interactive and intuitive manner, with any required data processing handled independently. All visualization components are accessible by an API and bindings enabling their invocation, orchestration, and integration in diverse processing environments and workflows. Therefore, they can be embedded in existing applications or be used as standalone assets in an exploratory data setting.

The following components constitute the SDL-Vis module (see Section 3.3 for more details on each component):

- **Visual Analytics Engine.** This component implements and instantiates the *visual analytics model*, which is a theoretical framework that structures and defines the interplay of data access, mining algorithms and interactive visualizations. It acts as an intermediate layer between the interactive visualizations and the lower layers of the SmartDataLake software stack, i.e., SDL-Virt and SDL-HIN. It abstracts data access for visualization and provides

vertical optimization functions, exposing its functionality in the form of a *visualization API*. This API exposes a RESTful service catalogue, as well as the actual Web services of all supported visualization artifacts.

- **Visual Explorer.** This component provides interactive visualizations, integrating the user into the process of knowledge generation. In particular, the *Graph Visualization* component provides a collection of fit-for-purpose scalable interactive visualizations for large-scale HINs. It enables the exploration of both the structure and attributes/properties of nodes, implementing various visualization and layout algorithms efficiently representing large-scale HIN data assets, ensuring high-performance and interactivity. Moreover, the *Spatial and Temporal Visualization* component provides a collection of fit-for-purpose scalable interactive visualizations for large-scale *spatial and/or temporal data assets*. It builds upon the map-based visualization of data assets (*i.e., layers, pan, zoom*), analysis results, geospatial concepts, and temporal aspects, gracefully addressing scalability challenges to provide a seamless and interactive exploratory experience for the user. Finally, the *Data Profiling and HIN Tuning GUIs* provide support for feature exploration and parameter tuning over heterogeneous data assets and large-scale HINs according to the interactive computing paradigm. Their purpose is to streamline exploratory data analysis, enabling the user to integrate domain-specific knowledge, formulate and test hypotheses about the data and actively improve the quality of results by incorporating the intuition and domain knowledge in the process.

3. SDL Platform Components

In this section, we introduce the individual software components that comprise the SDL platform and will be developed or extended during the project. For each component, we provide a short description, explain its role in the broader SDL platform, present its main functionalities, and describe its internal architecture.

3.1. SDL-Virt

The SDL-Virt module provides unified access to heterogeneous data in the data lake with the use of an SQL-like language. It enables access to *individual records*, as well as (both exact and approximate) *aggregate results*.

3.1.1. RAW

3.1.1.1. Short description

RAW is a system developed by RAW Labs. It permits to process raw data files to give their content a strong structure thanks to regular parsing but also ad hoc transformations of the data thanks to

its powerful expression language[1]. For example, it is possible to point it to a regular CSV file but then keep processing it before exposing its data: type certain fields as numbers or dates, skip records not matching a certain condition, split a string into more tokens (leads to a nested collection), parse another one with a regular expression, etc. Eventually the data exposes a more sophisticated structure matching the expected data type.

3.1.1.2. Role in SDL

The role of RAW is to provide flexible access and parsing of *raw data files* to turn them into *structured data* to be further processed by the user query. This first level processing of a data file is expressed using RAW's expression language and requires some basic understanding of the input file, in particular how the information was encoded and what kind of peculiarities to expect. The properly structured data produced by that processing is logically exposed as a RAW view which permits Proteus to query it as if it was a regular, well-typed table.

Requests to access and process raw data come from Proteus as part of its execution of the user query. Since RAW's expression language also supports filters and projections, Proteus' query planner may optimize the execution by deciding to push such operations to RAW, in which case, after parsing the file as the view specifies it, RAW would execute itself those filters and projections before outputting data.

RAW is also used to export structured data to the Storage Manager.

3.1.1.3. Main functionalities

RAW is an existing component that will be further extended and enhanced in the SmartDataLake project. Below, we outline its currently existing features and the ones that are planned to be developed during the project.

Existing functionalities:

- A flexible and powerful expression/query language (SQL-like).
- Parsers for common file formats like JSON, XML or CSV.
- Storage backends to read/write data (S3, Dropbox, plain file systems).
- REST APIs to execute a query and retrieve results.
- Python APIs to trigger the execution of a query and retrieve its results.

Planned functionalities:

- API to integrate with Proteus: similar to existing Python one but in the programming language used by Proteus during query preparation (C++) and execution (C).
- APIs to expose capabilities and operator cost to the query planner.
- Interfacing with the storage manager for both reading and writing.
- Support of file formats used by use cases in the project.

[1] https://www.raw-labs.com/raw-language-features

3.1.1.4. Architecture

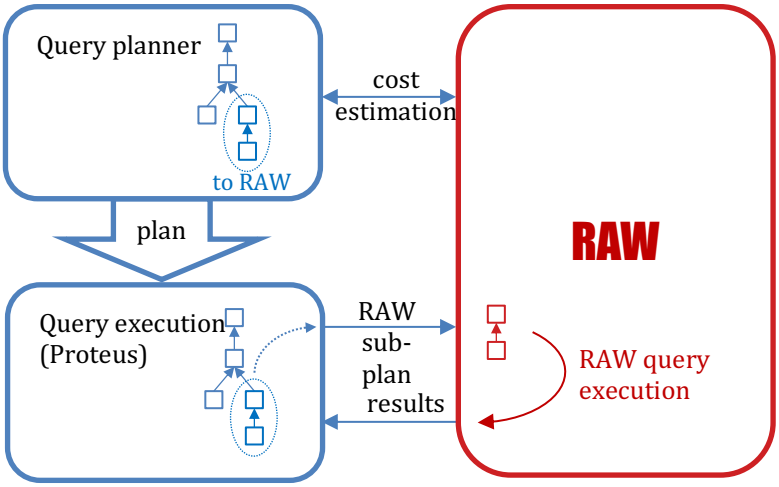

Figure 3: Internal architecture of RAW.

The Query Planner produces a query plan of which a fraction is to be executed by RAW: access to raw data files at a minimum, but possibly some filter/projection operators. The decision of the Query Planner to push those operators to RAW takes place during optimization and involves interfacing with the Query Planner so that it is aware of the capabilities of RAW and can estimate the cost of a given operator if run by RAW. Then during the execution of the query plan by Proteus, the fraction of the plan to be executed by RAW is turned into a RAW expression and passed to the RAW API. The RAW API returns a stream of data for Proteus to execute the user query.

In case RAW should export its data to storage instead of returning it as a byte stream to Proteus, the API remains the same and the interaction with Proteus is similar. Proteus triggers the execution of the RAW query with a specific parameter to instruct it to save results in a given location.

3.1.2. Proteus

3.1.2.1. Short description

Proteus [Karpathiotakis2016, Chrysogelos2019] is the query execution engine which is used to run queries for SDL. It provides the implementation for various SQL operators that will be used in running queries. Proteus uses data virtualization for query execution over heterogeneous data and uses a plugin architecture to support different data formats. It uses runtime code generation with LLVM[2] to support execution of query operators over heterogeneous hardware.

[2] https://llvm.org/

3.1.2.2. Role in SDL

Proteus will be the query execution engine for SQL queries in SDL. All SQL queries, both approximate and exact, will be run on Proteus. Proteus generates custom code for running query operators on heterogeneous hardware.

The Query Planner component interfaces with Proteus to provide the query execution plan. Proteus uses the Resource Manager to allocate the resources for running different operators on the query. The Storage Manager and RAW are used to access data needed to process the queries.

3.1.2.3. Main functionalities

Proteus is an existing component that will be further extended and enhanced in SDL. Below, we outline its currently existing features and the ones that are planned to be developed during the project.

Existing functionalities:

- Query execution over both CPUs and GPUs.
- Query execution over heterogeneous data using a plugin architecture.

Planned functionalities:

- *Enhanced data access*: we plan to add the following data access features to Proteus:
    - Data access from RAW: RAW provides efficient access to different raw data formats. We plan on adding support for data access over RAW. We also plan on adding support in Proteus to push certain operators like selections and projections to reduce the amount of data that needs to be processed by Proteus.
    - Heterogeneous data access: The input data could be present at different levels of hierarchy. In particular, data may need to be accessed from cloud storage services. We plan to add support for optimally accessing data stored at different storage hierarchies including cloud storage.
- *Scale out query processing*: Proteus currently runs on a single machine only. For processing queries across multiple nodes, we plan on using the Resource Manager to allocate resources to run the query. We plan to add support for distributing input data and intermediate results across nodes. We also plan on implementing support for running operators in parallel across multiple nodes, as well as for synchronization to check when an operator finishes and releasing resources accordingly.
- *Approximate query processing*. We plan to add *approximate query processing* capabilities. This involves changes at different components, including both the query planner and the Proteus query execution engine. In terms of the execution engine, the following functionality will be offered:
    - New physical and logical operators to support approximate queries. This includes: (a) operators for query execution, and (b) operators for generating, storing, and maintaining the synopses.

- A synopsis catalogue, which can be used by the Query Planner to decide which synopses are the best for each query, and how to optimize for future queries.
- A statistics maintenance component, which will maintain sufficient data for empowering the self-tuning nature of the Query Planner. Among others, these statistics include access patterns of the data and access paths, access cost, and information about data distribution.

Notice that Proteus itself will not make decisions about planning or about creating and maintaining synopses. Instead, the self-tuning and planning nature of the query engine will be taken within the Query Planner (see Section 3.1.3). Then, Proteus will implement these decisions.

3.1.2.4. Architecture

Proteus takes a query plan as input from the Query Planner. Proteus generates code at runtime based on the operators it has to execute and the hardware on which the operators are to be executed (either CPU or GPU). It supports intra-operator parallelism allowing a single operator to run on multiple threads in parallel, as well as pipelining of query operators. Proteus uses a plugin architecture for reading inputs in heterogeneous formats.

Proteus accesses its input data from the Storage Manager and RAW.

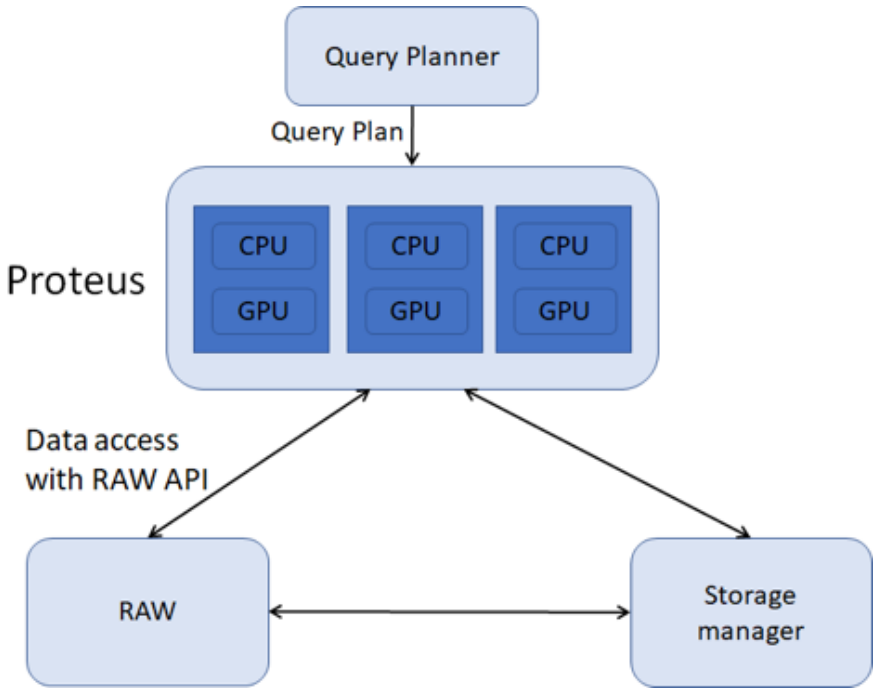

Figure 4: Architecture of Proteus.

3.1.3. Query Planner

3.1.3.1. Short description

The Query Planner component of SDL takes the SQL queries as input and generates optimized physical query plans for Proteus. The plan generated by the Query Planner specifies the order of operators, the degree of parallelism for each operator, as well as the hardware to run the operators on. For approximate queries, the Query Planner also specifies which synopsis to access and the sampling rate for underlying relations.

3.1.3.2. Role in SDL

The Query Planner parses the SQL query provided to it and generates an optimized query plan for the Proteus execution engine. It interfaces with the Storage Manager and RAW to estimate data access costs and get statistics on the underlying data. When generating query plans it also takes into account the heterogeneous nature of the data, as well as the heterogeneous hardware on which the query is to be executed. The Query Planner component hides the complexity of the underlying heterogeneous hardware.

3.1.3.3. Main functionalities

The Query Planner is an existing component that will be further extended and enhanced in SDL. Below, we outline its currently existing features and the ones that are planned to be developed during the project.

Existing functionalities:

- *Physical query plan generation for SQL queries:* The Query Planner generates a physical query plan for the given SQL query to run on a single node.
- *Query plan generation for heterogeneous architecture:* The SQL query plan specifies which part of query plan runs on which hardware.

Planned functionalities:

- *Query optimization for heterogeneous architecture*: The Query Planner will be able to optimize the query plan based on the underlying hardware, the data statistics and cost estimations.
- *Data access*: We plan to add the following data access features to the Query Planner:
    - RAW provides efficient access to different raw data formats. Pushing operators like selections and projections to RAW when accessing raw data would be useful since it could reduce the size of data that needs to be processed in the future. Also, operators like nesting and unnesting can be pushed to RAW. We plan on using the Query Planner to determine the operators to be pushed down to RAW to optimize the data access.

- The data to be accessed for query evaluation may reside at different storage hierarchy levels as decided by the storage manager. We plan on using the access cost of different storage hierarchy levels to optimize the query plan.

- *Scale out query processing*. The Query Planner currently generates query plans for a single machine. To this end, the following extensions are planned:
  - To support scale out computing for SDL, we plan to add support for generating query plans that run across multiple nodes in a distributed manner.
  - In a scale out setting, data may be replicated across multiple nodes and data access cost of all replicas may not be the same. There is some cost involved in transferring intermediate results as well. The query optimization will take into account the data transfer and access cost when running in a distributed setting.
  - All nodes used in the SDL may not have the same amount of resources. Some nodes may have multiple sockets and powerful CPUs while others may have powerful GPUs. Depending on the type of query it may be useful to use more of one type of nodes vs others. We plan on supporting query plan generation for nodes with different resources.
- *Approximate query processing*. We plan to add approximate query processing capabilities. This involves changes in the Query Planner and in the Proteus query execution engine. In terms of the Query Planner, the following functionality will be offered:
  - Logical planning: Generating approximation-aware logical plans, and choosing the best one, according to an optimization strategy (e.g., short term – reduce response time for the current query, or long-term – increase the performance of the next 100 queries).
  - Physical planning: Generating a physical plan that includes the synopsis generation step and sending that to the query engine for further execution.
  - Self-tuning: Deciding which synopses to generate for enabling better plans for the current and future queries.

These functionalities will be achieved with close coordination with the Storage Manager, which regulates access to the storage hierarchy, i.e., it decides where each data segment should be stored. Our first results on approximate query processing are presented in [Olma2019].

3.1.3.4. Architecture

The input to the Query Planner is an SQL query. The query may be an exact query or an approximate query with specified error-bounds. The Query Planner will generate a query plan using the data indexes, data statistics and the hardware information it has.

When generating a query plan for an approximate query, the planner also checks the summaries. If summaries that can answer queries with desired error bounds are present, it generates a query plan that uses summaries. Otherwise, the Query Planner generates a plan using sampling operators. The planner may also decide to generate summaries for the approximate query if, based

on past history, it decides that generating such a plan would be beneficial for current and future queries.

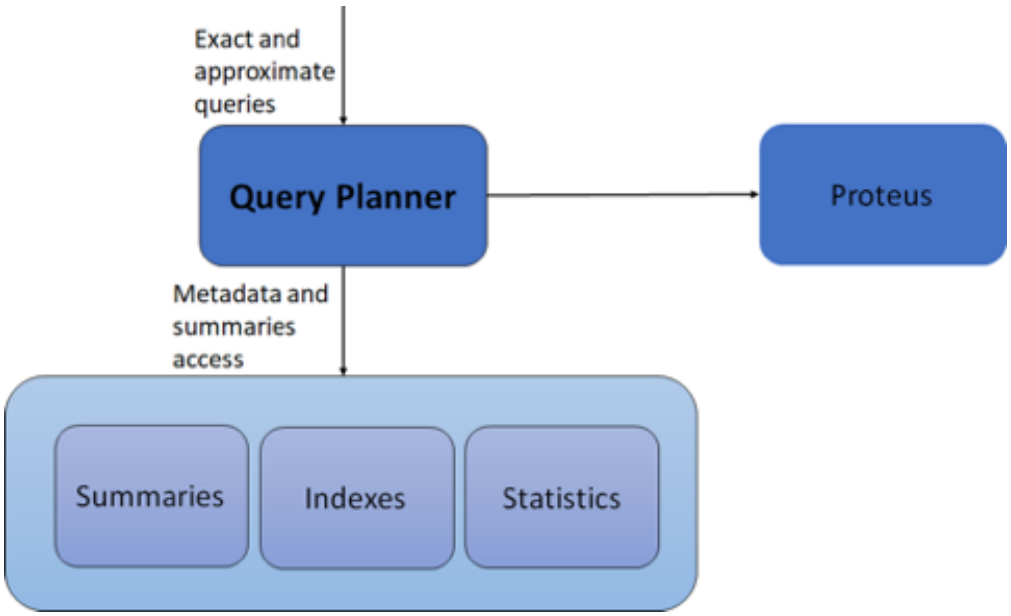

Figure 5: Architecture of the Query Planner.

3.1.4. Resource Manager

3.1.4.1. Short description

The Resource Manager will enable the specification, implementation and enforcement of resource allocation policies at runtime. These policies will include information coming both from the workload and from the availability of the underlying resources, whereas they will consider the capabilities of the SDL infrastructure to elastically scale across all the possible dimensions. Such dimensions include the storage hierarchy, the processing capabilities, as well as access to shared resources, like the network bandwidth.

3.1.4.2. Role in SDL

The highly heterogeneous nature of SDL, spanning to data formats, storage hierarchy and approximate/exact query operators in the query plans, increases the dimensionality of the resource planning problem for query execution in the cloud. For this reason, SDL will include a flexible Resource Manager which will allow fine-grained resource management by moving resources, like CPUs, memory and network bandwidth across different tasks running in parallel on top of the SDL multi-tenant software platform. The Resource Manager will draw inspiration from other existing frameworks offering similar functionality, like Kubernetes and Apache Yarn and Mesos. However, its scope in SDL is larger, as it requires the tight collaboration with the query

engine in order to make optimal decisions on how to achieve optimal distribution of resources across different queries. Moreover, it requires collaboration with the Storage Manager, as queries might need different parts of the storage hierarchy for intermediate results, which might also have to be moved across the different layers at query runtime. Finally, the Resource Manager will provide isolation to the execution of applications by leveraging technologies like virtual machines and containers.

3.1.4.3. Main functionalities

The following functionalities are planned for the Resource Manager:

- *Resource isolation*: The Resource Manager will enable task execution in isolation, but at the same time it will allow tasks to communicate efficiently by trading resources without breaking the isolation guarantees. For instance, task communication running on the same server can take place by exchanging parts of the memory that belong to a globally assigned buffer.
- *Policy management*: The Resource Manager will provide an API that allows system administrators to create policies that mediate access to resources and hotplug resources based on their availability and the workload properties. These policies will consider all the different dimensions of heterogeneity that SDL has to handle.
- *Resource orchestration*: The resource orchestrator will consider the rules specified by the policies associated with each task running on the platform and will decide and enforce the optimal schedule. SDL will consider different scheduling approaches based on the metric that should be optimized every time, like throughput, response time, resource utilization, etc.

3.1.4.4. Architecture

The Resource Manager will offer a unified API for accessing different resources. Moreover, it will provide functionality for creating and storing policies, leveraged by the resource orchestrator, which will be responsible for distributing resources to the different tenants (tasks) of the system. Finally, resource utilization will be monitored by the resource monitor, which will provide feedback to the orchestrator in case more resources are needed by a task, and vice versa.

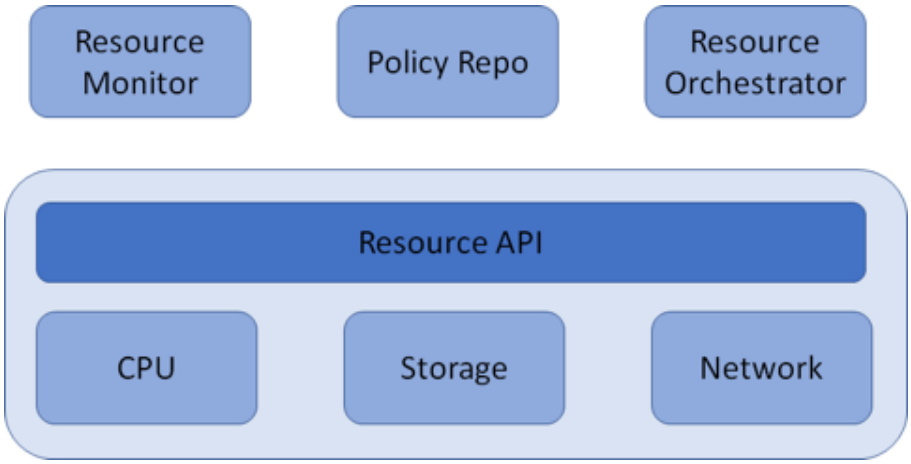

Figure 6: Architecture of the Resource Manager.

3.1.5. Storage Manager

3.1.5.1. Short description

The Storage Manager will be a self-tuning and elastic component managing the storage hierarchy of SDL. It will mediate access to data with respect to different software and hardware parameters. On the software side, the Storage Manager will have to support access to raw data, intermediate representations, including transformations in different formats (e.g., binary) and caches, as well as the synopses used in the approximate query processing context. On the hardware side, the storage hierarchy may include a combination of resources, each with different access parameters, that are accessed either locally or remotely. Indicatively, the storage hierarchy can include main memory (RAM), different types of persistent storage ranging from HDD to SSD/NVMe, as well as network storage.

Bearing in mind the storage heterogeneity with respect to the different requirements of the applications running on top of the data lake, the storage manager will also rearrange data across the different layers of the hierarchy in order to optimize data access times.

3.1.5.2. Role in SDL

All data accesses that take place through the data virtualization layer will pass through the Storage Manager. Specifically, components like the Query Planner and the query execution engines will use the Storage Manager abstraction to have unified access to the data, independent of the actual storage layer and data representation.

The remaining components of the reference architecture may also use the Storage Manager as an object store, thereby directly exploiting its storage hierarchy abstractions. For example, they can store configuration files, or big data in arbitrary, non-queryable and non-indexable format (videos,

images, etc.). Storing these files directly through the Storage Manager API will be more efficient than storing them via the data API, as it will skip one redirection.

3.1.5.3. Main functionalities

The Storage Manager will offer two interfaces:

- A URL-based read/write interface, which can be used from other components to access or modify the data. The data can be either the actual data that we want to query, or auxiliary data like synopses. In the following, we use the term objects to refer to all possible data types;
- An interface for creating a new URL with a set of properties, i.e., expected data size, ways that this handler will be used (e.g., how frequent will be the access on this URL, sequential reads or random accesses). The Storage Manager will then decide where to allocate space for this URL, based on a parameterized cost model.

Internally, the Storage Manager will also satisfy the requirements of elasticity (adding and removing resources, as instructed by the resource manager), self-tuning (deciding to change storage layers of some objects in order to increase throughput), and fault-tolerance (ensuring that a failure does not lead to loss of critical data).

3.1.5.4. Architecture

The Storage Manager exposes a unified API to the other components which allows them to access data transparently. Then, based on the device where each part of the data accessed is located, the Storage Manager implements the corresponding driver. Drivers are initialized by each application, since they might require different properties, based on the type of data accesses. Such configuration parameters will include the page size, as well as device-specific features such as NVMe over fabric which can be applied in specific cases to increase performance, however prior knowledge on the workload is required and therefore the Storage Manager cannot make a decision on its own.

In addition to data access, the Storage Manager also provides a component that monitors the performance of the storage accesses and keeps statistics so that it can rearrange data throughout the whole data lake. The policies for the latter are kept and implemented by the data rebalancer.

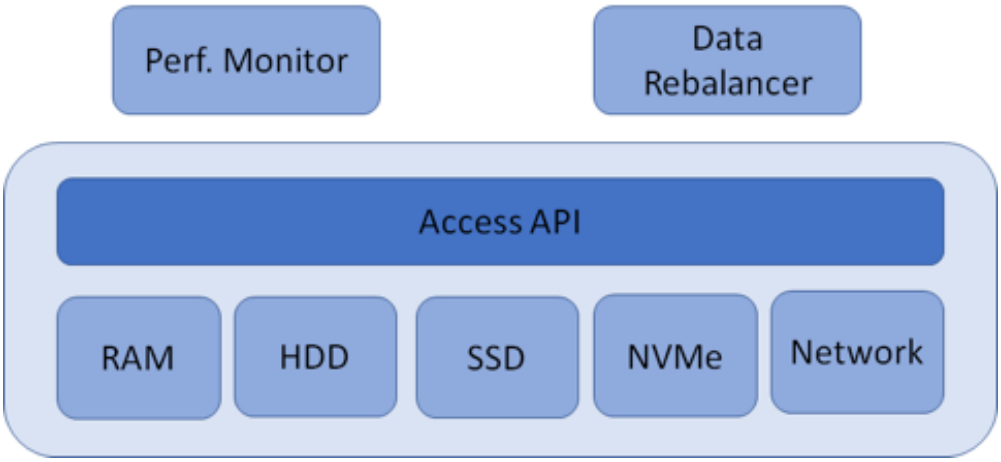

Figure 7: Architecture of the Storage Manager.

3.2. SDL-HIN

SDL-HIN provides functionalities for exploring and analyzing HIN data assets. In particular, it supports similarity search over entity profiles, entity resolution and ranking, link prediction between entities, detection of entity groups and communities, and management of changes. SDL-HIN can be deployed and used on its own (see Section 4.3.2) or alongside SDL-Virt and/or SDL-Vis (see Section 4.3.4). Moreover, its services can be invoked independently or as parts of broader data analysis workflows (see Sections 4.4.3.2, 4.4.3.3 and 4.4.3.4).

Next, we describe each of the components constituting this module.

3.2.1. HIN Engine

3.2.1.1. Short description

HIN Engine is responsible for managing HIN-induced graph and providing *graph storage, maintenance and querying* capabilities to components in SDL-HIN and SDL-Vis modules, where required.

3.2.1.2. Role in SDL

SDL uses HINs to represent different types of relations among different types of entities derived from the contents of the underlying data lake. It is natural to model the HIN itself as a graph. Such HIN-induced graph can become very large when constructed over massive data lakes. Most of the components in SDL-HIN and SDL-Vis require efficient reasoning both over the HIN graph and over HIN's underlying data assets. HIN Engine component is designed to enable the former while the latter is delegated to SDL-Virt through its data API.

3.2.1.3. Main functionalities

HIN Engine manages the HIN-induced graph and provides core components in SDL-HIN and SDL-Vis with homogeneous access to HIN data assets queried through the data API of SDL-Virt or provided directly as input (i.e., passthrough).

As such, it is responsible for constructing, maintaining, and providing efficient access to a meta-graph that corresponds to a HIN. To achieve this, HIN Engine sets up various APIs (graph navigation, auxiliary manipulation, visualization, and passthrough) as needed by both SDL-HIN and SDL-Vis components.

HIN Engine also ensures that the corresponding API calls are processed in a performant manner by employing efficient graph storage coupled with effective graph query optimization. HIN Engine guarantees that all assets follow the same required model, serves intra-/inter-analysis pipelines (e.g., interim results, logs), lowers resource utilization in case of analysis workflows with identical target assets (typical for interactive computing), and affords a degree of independence in a production setting from the underlying data lake system (e.g., in the case that a pre-existing graph needs to be used across analysis workflows as is).

3.2.1.4. Architecture

Figure 8 illustrates the main components of HIN Engine as outlined below:

- *Storage*: HIN Engine uses a collection of specialized data structures to store the HIN graph both on volatile (e.g., memory) and non-volatile (e.g., SSD/HDD) storage. These data structures are tailored to enable efficient primitive operations on graphs such as addition or removal of nodes/edges, edge traversals, reasoning over node/edge properties, and others. Additionally, various graph statistics and metrics are maintained in the catalogue which aids in query optimization. The catalogue module interacts with the Change Manager component (see Section 3.2.4) to keep these statistics and metrics up-to-date.
- *Indexes*: Complex operations over HIN graph (e.g., non-trivial navigation, transitive closures, aggregation for visualization) are aided with a collection of purpose-built indexes over a HIN graph. Three types of indexes are distinguished: *visual*, *path*, and *metric* indexes. *Visual* indexes are specialized data structures tailored to support efficient execution of operations induced by SDL-Vis components such as graph aggregation, exploration, zooming and panning. *Path* indexes pre-materialize and efficiently store interesting paths such as *meta-paths* used in the HIN Miner component (see Section 3.2.3) to express different ways in which various entities are related to each other in the HIN. Finally, *metric* indexes are designed to facilitate the maintenance of various graph metrics by the Change Manager component and used throughout SDL-HIN and SDL-Vis to aid in HIN analysis.
- *APIs*: Four categories of APIs grouped by their respective functionality are maintained by the HIN Engine: *navigational*, *visual*, *auxiliary*, and *passthrough*. These APIs interact with corresponding modules in the SDL platform to provide functionality as required. For example, visual API (Vis) deals mainly with SDL-Vis module to perform data-centric tasks in HIN visualization.

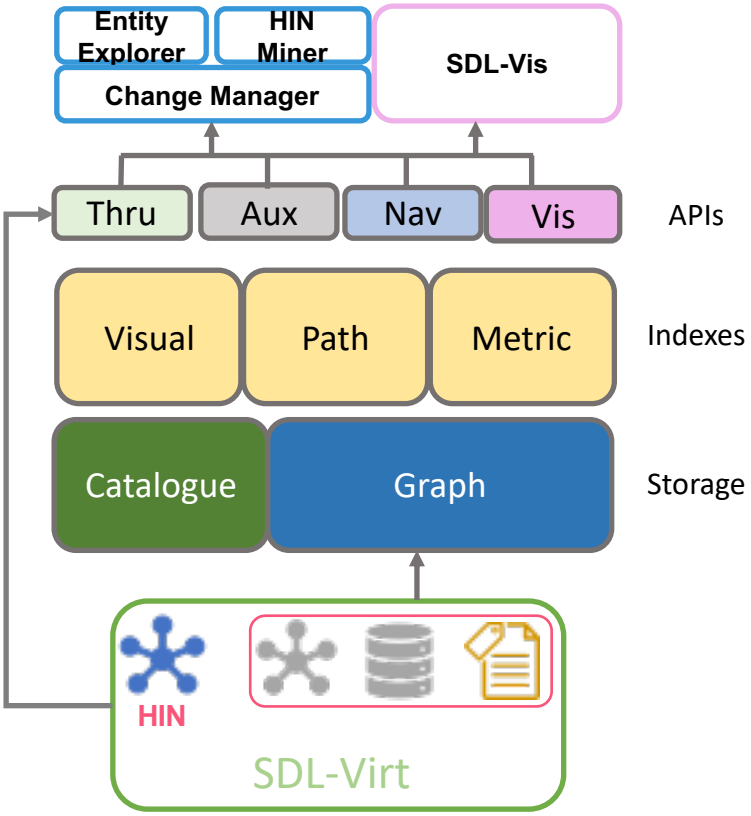

Figure 8: Main components of HIN Engine.

3.2.2. Entity Explorer

3.2.2.1. Short description

Entity Explorer operates over *collections of entity profiles* and provides functionalities for *similarity search* and *entity resolution*.

Entity profiles. Entities correspond to *nodes* in the HIN and can be of multiple *types*. For example, an entity can be a company, an organization, a product, a project, a publication, an author, etc. An *entity profile* is represented by *a set of key-value pairs*, where *key* denotes some attribute of the entity and *value* is the corresponding value. To handle entities of different types and to deal with the heterogeneity of the input sources, we make no assumptions about the keys, i.e., what each key means or whether different sources may use the same keys to refer to the same attributes. Thus, in many cases, the keys may be of no particular usefulness, which means that an entity profile practically becomes a list of values. Particular emphasis is given on lists of values where the values are *strings* or *sets of strings*. This is sufficiently flexible and abstract to cover a variety of scenarios, as indicated by the following examples:

- *Representing a company by its name*: the profile comprises a single string (in other words, a list with a single entry of type string).

- *Representing a company by a list of its products*: the profile comprises a list of strings, each one denoting the name of a product.
- *Representing a company by a list of its product characteristics*: the profile comprises a list of sets, where each set is a collection of tags or keywords describing the respective product.

Other similar examples include, for instance, representing a researcher by her name or by a list of the titles of her publications or by a list of sets containing the keywords of her publications.

Similarity search. Given a collection of entity profiles, *similarity search* refers to identifying *pairs of similar profiles* [Wandelt2014, Yu2016, Jiang2014, Mann2016, Fier2018]. In particular, we consider three variants of this operation:

- *Search* – Given a collection of entity profiles D and a query entity profile E, find all profiles in D that are similar to E.
- *Join* – Given two collections of entity profiles D_1 and D_2, find all pairs of entity profiles (E_i, E_j), where E_i in D_1 and E_j in D_2, such that E_i is similar to E_j.
- *Self-join* – Given a collection of entity profiles D, find all pairs of entity profiles (E_i, E_j), where E_i, E_j in D and $j > i$, such that E_i is similar to E_j.

The similarity is based on a given *similarity measure* and a *condition*. The similarity measure is typically a set or string similarity measure, such as Jaccard similarity or Edit Distance. The condition can be *threshold-based*, e.g., find all pairs with similarity at least 0.8, or *top-k*, e.g., find the 5 most similar entities to a given entity.

Entity resolution. Given a collection of entity profiles, *entity resolution* refers to identifying *pairs of entity profiles that correspond to the same real-world entity*. [Christophides2015, Dong2015, Getoor2012, Papadakis2016]. Entity resolution may be performed within a single collection (often referred to as *deduplication*) or across two different collection (often referred to as *linkage*). Compared to similarity search, instead of a similarity computation, *entity matching* is performed. The exact method used for entity matching may differ significantly depending on the application. For example, a simple but usually not very effective option is to consider as matching those entity profiles that are very similar to each other. Usually, application-specific *matching rules* may be provided, indicating when two entity profiles constitute a match, or *ML classifiers* are applied that have been previously trained on provided examples of matching and non-matching profiles.

3.2.2.2. Role in SDL

A data lake contains information about different types of entities coming from multiple, heterogeneous data sources. The role of Entity Explorer is to provide functionalities for discovering entity profiles that are similar to each other or that refer to the same real-world entity. Both of these operations are fundamental for further analysis and knowledge extraction. Entity resolution allows to deduplicate a data source or a collection of data sources, which would otherwise affect the accuracy of the results of queries or analyses over it. Moreover, it allows to link together segments of information for the same entity in order to derive more complete entity

profiles. Similarity search and join is an important operator for other tasks, such as link prediction and clustering.

Entity Explorer interfaces with the Query Planner of SDL-Virt to obtain virtualized and efficient access to the contents of the underlying data lake. Its output is used by HIN Miner that provides further analyses on the entities in the HIN. It also interacts with Change Manager for detecting and handling changes. Finally, it interfaces with SDL-Vis for visual parameter tuning and visualization of the results.

3.2.2.3. Main functionalities

The following list outlines the planned functionalities for this component:

- *Similarity search*: Given a query entity profile and a collection of entity profiles, this operation retrieves all entity profiles in the collection that are similar to the query. More specifically, the following variants will be supported:
    - *Threshold-based search*: a similarity threshold is given as input, and all entity profiles with similarity to the query higher than this threshold are retrieved.
    - *K-Nearest Neighbor (kNN) search*: a fixed number of k entity profiles are retrieved, having the highest similarity to the query.
- *Similarity join*: Given two collections of entity profiles, this operation identifies all pairs of entity profiles that are similar to each other. *Self-join* is also supported, referring to the case where all pairs of similar entity profiles within the same collection are identified. More specifically, the following variants are addressed:
    - *Threshold-based join*: a similarity threshold is given as input, and all pairs with similarity higher than threshold are retrieved.
    - *kNN join*: for each entity profile in the first collection, its k-nearest neighbors from the second collection are retrieved.
    - *k-closest pairs*: a fixed number of k pairs are retrieved, being the ones with the highest similarity scores.
- *Rule-based entity matching*: Given two collections of entity profiles and a set of matching rules, all matching pairs are identified.
- *ML-based entity matching*: Given two collections of entity profiles and a set of training examples of matching and non-matching entities, an ML model is trained and then applied to detect matching entity profiles.

3.2.2.4. Architecture

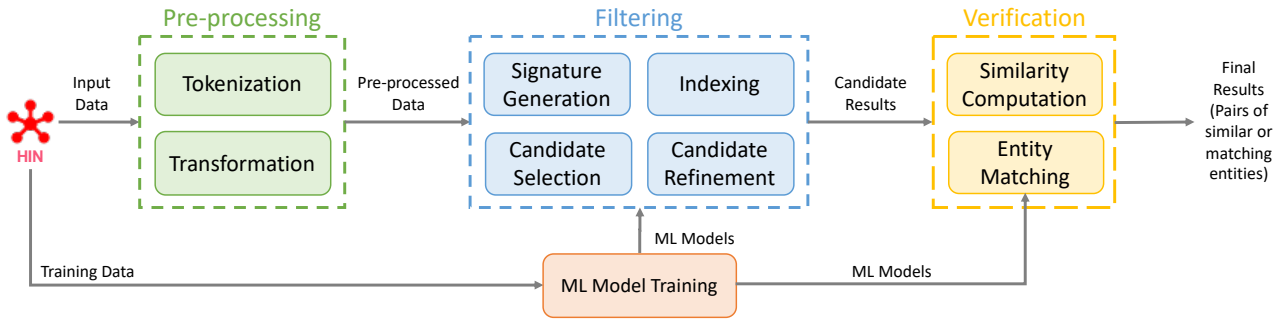

Figure 9: Main components of Entity Explorer.

Figure 9 illustrates the main components of Entity Explorer for performing similarity search and entity resolution. Our design follows a common framework for both of these two operations, based on a *filter-verification* approach. The processing involves three main stages, as described below:

- *Pre-processing*. This stage involves components that tokenize and transform the input so that it can be used by the subsequent algorithms that perform the similarity search and entity resolution tasks.
- *Filtering*. This stage is the core part of the similarity search and entity resolution algorithms and is essential for the efficiency and scalability of the process. Its purpose is to avoid the inherent quadratic complexity of both tasks; instead of performing all pairwise comparisons between the input entities, it aims at identifying a set of candidates so that comparisons are eventually performed only between those pairs that are likely to be similar or to constitute a match. Signature generation employs various algorithms to extract certain information from each entity profile, which is then used to select an initial pool of candidates. Additional filters may be applied to further reduce the number of candidates. Appropriate indices are constructed and used by the filtering algorithms to guide the candidate selection and refinement processes.
- *Verification*. At this stage, the surviving candidates are verified. For similarity search, the similarity function is invoked to compute the exact similarity score of each candidate pair. For entity resolution, an entity matcher, based either on matching rules or ML models, is invoked to determine whether a candidate pair constitutes a match or not.

3.2.3. HIN Miner

3.2.3.1. Short description

HIN Miner operates over a *HIN* and provides functionalities for *entity ranking*, *link prediction* and *community detection*. These operations take into consideration *meta-paths*, to exploit the rich

semantics in HINs, and exploit features that may be extracted directly from the HIN or via *HIN embeddings*. We briefly describe these concepts and operations below.

Meta-paths. The high expressiveness of HINs lies in the fact that they allow to model different types of relationships between different types of entities [Shi2017]. Meta-paths capture these rich semantics by expressing different ways in which various entities are related [Meng2015]. A meta-path is a path defined on the schema of the HIN, i.e., comprising node and edge types. For example, the meta-path Company – Supplier – Company relates together companies having the same supplier, while the meta-path Company – PointOfSales – Location – PointOfSales – Company relates companies having points of sale in the same location. Thus, selecting one or more meta-paths allows the user to explore and analyze the HIN from different perspectives.

HIN embedding. A HIN embedding is a transformation that maps a HIN to an n-dimensional space, representing each entity (node) as an n-dimensional vector, in such a way that the structure of the HIN is preserved as much as possible [Chen2017, Shi2019, Huang2017]. This means that entities that are close to each other in the HIN should be mapped to points that are also close to each other in the transformed space. This representation can then be used by various algorithms for exploration and analysis.

Entity ranking. Given a HIN, entity ranking refers to assigning scores to entities to measure the importance of each entity with respect to others [Soulier2013, Li2014]. This could be measured, for instance, by computing the frequency of certain relationships between certain entities, e.g., how many times a certain company is mentioned in news articles. Often, more complex relations need to be considered (i.e., *paths* instead of *edges*); for instance, how many times a researcher has published a paper in a certain conference. Moreover, the importance of an entity has an impact to the importance of other entities in its neighborhood (e.g., a news article may be considered as more important if it concerns important companies, and vice versa). This requires *random walk* algorithms for ranking the importance of entities in the graph. Moreover, the ranking can be *absolute* or *relative*. In the latter case, the ranking is computed with respect to an initial context or preferences (e.g., find the top authors with respect to conferences related to Big Data).

Link prediction. Given a HIN, this operation aims at predicting new links between entities [Chen2018, Martinez2017, Yang2012]. These may represent links that exist but are not available in the current data or links that are considered as likely to be formed in the future. Since a HIN comprises different types of relations, meta-paths are important in link prediction. Specifically, different meta-paths can be used as features for the prediction. Moreover, different models need to be trained to predict relations for different meta-paths.

Community detection. Given a HIN, this operation identifies communities (i.e., groups) of entities that are related according to certain criteria [Zhan2017, Aggarwal2011, Boden2014]. Again, meta-paths play an important role here, because they are essentially the criteria that determine how the *proximity* of two entities is defined and thus how the groups are formed. We use the term *community detection* to refer to the task of discovering all communities in a HIN, while the term *community search* to refer to the task of discovering the community that a given entity belongs to.

3.2.3.2. Role in SDL

SDL uses HINs to represent different types of relations among different types of entities derived from the contents of the underlying data lake. HIN Miner provides the means to analyze such HINs and extract additional knowledge. Specifically, it focuses on three main functionalities: (a) ranking entities according to their position in the network, (b) predicting links that may be currently missing or are likely to be formed in the future, and (c) grouping together related entities according to different types of relations.

HIN Miner relies on Entity Explorer for computing similarities between entity profiles. It also interacts with Change Manager for detecting and handling changes in the data. Finally, it interfaces with SDL-Vis for visual parameter tuning and visualization of the results.

3.2.3.3. Main functionalities

The following list outlines the planned functionalities for this component:

- *Entity ranking*: Given a HIN, this operation assigns ranking scores to entities (HIN nodes), taking the network structure into account. The ranking can be *absolute* or *relative*. In the latter case, an additional input is provided which denotes the user's preferences or interest with respect to certain entities. This is taken into account in order to rank entities in the network with respect to those ones preferred by the user. Moreover, the ranking can be *global* or *top-k*. In the former case, a ranking score is assigned to each entity in the HIN, while in the latter case, only the top-k entities are identified and returned.
- *Link prediction*: This operation takes as input a HIN and the type of relation to be predicted. Based on a training dataset including positive and negative examples, an ML model is trained and is then applied to predict new links between nodes in the network. A query node may be indicated, in which case the links for this particular node are predicted, or the model can be executed to predict new links for all nodes in the HIN.
- *Community detection*: This operation takes as input a HIN and one or more meta-paths that indicate which type(s) of relation(s) between entities are of interest, i.e., indicate that entities relate or interact with each other. Based on these, it identifies communities in the HIN, i.e., groups of entities that are closely related to each other. Depending on the employed algorithm, these communities may have a hierarchical structure, i.e., a larger community may be further partitioned into smaller communities. Moreover, two modes of operation are distinguished: *community search* and *community detection*. The former refers to identifying the community that a particular entity belongs to, while the latter refers to identifying all communities in the HIN.

3.2.3.4. Architecture

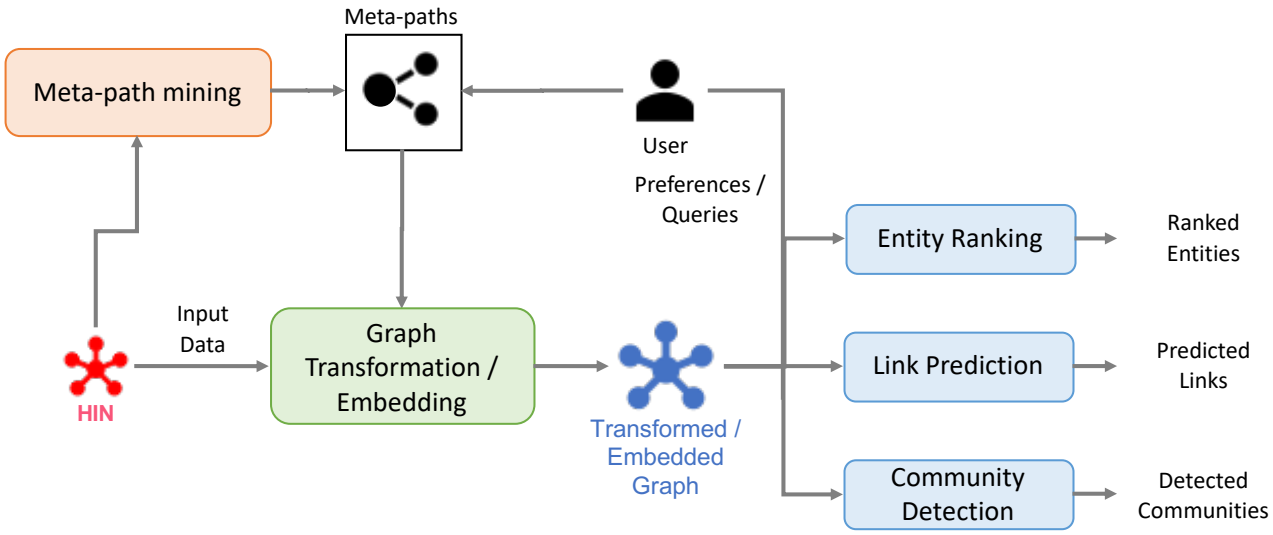

Figure 10: Main components of HIN Miner.

Figure 10 illustrates the main components of HIN Miner for performing entity ranking, link prediction and community detection, as outlined below:

- *Meta-path mining*. This is an optional component that automatically discovers and recommends interesting meta-paths in the HIN.
- *Graph transformation / embedding*. Given a set of meta-paths, either selected automatically by the meta-path mining component or provided explicitly by the user, this component transforms the original HIN to a new representation that facilitates further processing by the subsequent algorithms. This representation may be again a graph, e.g., with additional edges denoting the desired meta-paths, or an embedding of the original HIN in an n-dimensional vector space.
- *Entity ranking*. This component executes entity ranking algorithms on the transformed graph to compute entity scores and derive top-k lists of entities. The process takes into consideration any user preferences, if provided, in order to provide rankings that are personalized.
- *Link prediction*. This component operates on the transformed graph. It extracts features and trains ML models for predicting different types of links in the graph.
- *Community detection*. This component operates on the transformed graph. It executes clustering algorithms to detect communities and assign entities to them.

3.2.4. Change Manager

3.2.4.1. Short description

Change Manager provides support for detecting and managing changes in the HIN over time. Changes may occur for several reasons; in particular, new entities may appear (new HIN nodes), new relations between entities may be established (new HIN edges), or the characteristics of existing entities and relations may change (updates on existing node properties or edge weights). When such changes occur, the goal is to detect which previous results are affected and attempt to update them incrementally.

3.2.4.2. Role in SDL

Typically, the contents of a data lake are not static but are rather updated in a periodic or ad hoc basis. The role of Change Manager is to monitor changes in a given HIN that is being used or has been used for a certain analysis, determine whether these changes may affect the previous results, and accordingly trigger the respective component with the aim to update the results incrementally.

Change Manager takes the current and previous version of a HIN as input, as well as the previous analysis results, and interacts with Entity Explorer and HIN Miner to update the results.

3.2.4.3. Main functionalities

The following list outlines the planned functionalities for this component. All listed operations are triggered when changes in entity profiles occur.

- *Update similar entities*: Given a previously computed set of similar entity profiles, this operation takes into consideration the changes in the input entity profiles to determine how the previous results are affected, i.e., detect pairs that are no longer similar, as well as new pairs that now become similar.
- *Update entity rankings*: Given a previously computed ranked list of entities, this operation takes into consideration the changes in entity profiles to determine whether and how the ranking of entities has changed.
- *Update predicted links*: Given a previously computed set of predicted links in a HIN, this operation takes into consideration the changes in entity profiles to determine whether any previously predicted links are invalidated, as well as whether new predictions can be made.
- *Update communities*: Given a previously computed set of communities formed in a HIN, this operation takes into account the changes in entity profiles to determine which communities are affected and how, i.e., which communities may obtain new members or lose existing ones, and, accordingly, may be merged with another community or split into smaller communities.
- *Update graph metrics*: A number of graph metrics are used by SDL-HIN and SDL-Vis components to aid in HIN analysis, e.g., degree distributions, diameter, connectivity, centrality, and others. Given previously computed metrics, this operation considers

changes made to the HIN graph and identifies which changes and to which metrics are required. Further, necessary updates are executed efficiently avoiding full recomputation whenever possible.

3.2.4.4. Architecture

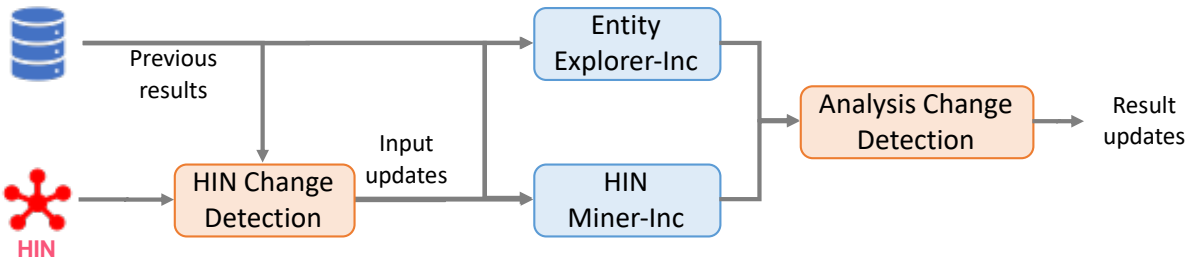

Figure 11: Main components of Change Manager.

Figure 11 illustrates the main components of Change Manager, as outlined below:

- *HIN change detection*. This component monitors the input HIN and notifies about occurring changes that may affect the results of previous analyses.
- *Entity Explorer-Inc*. This is an extension to Entity Explorer that enables incremental update of similarity search and join between entity profiles.
- *HIN Miner-Inc*. This is an extension to HIN Miner that enables incremental update of entity rankings, predicted links and detected communities.
- *Analysis change detection*. This component compares the updated results with the previous ones and notifies about any changes that have occurred.

3.3. SDL-Vis

SDL-Vis provides specialized visualization and interaction functionalities to represent the heterogenous data assets in SDL, which include graph, spatial, and temporal data. Furthermore, SDL-Vis includes the user into the *Visual Analytics Process* [Keim2010]. As shown in Figure 12, this human-in-the-loop process involves *Parameter Tuning* (PT) for SDL-HIN and *Data Profiling* (DP) for SDL-Virt, improving the quality of derived models and optimizing data access. Vice versa, by visualizing the results of these optimizations, it allows a more sophisticated knowledge generation on the human side of the visual analytics process.

To ensure reusability and separation of concerns, SDL-Vis is split into the *Visual Analytics Engine*, instantiating the *Visual Analytics Model* and enabling *Vertical Optimizations*, and the *Visual Explorer*, providing graphical user interfaces and therefore exposing the functionality of the visual analytics engine to the user.

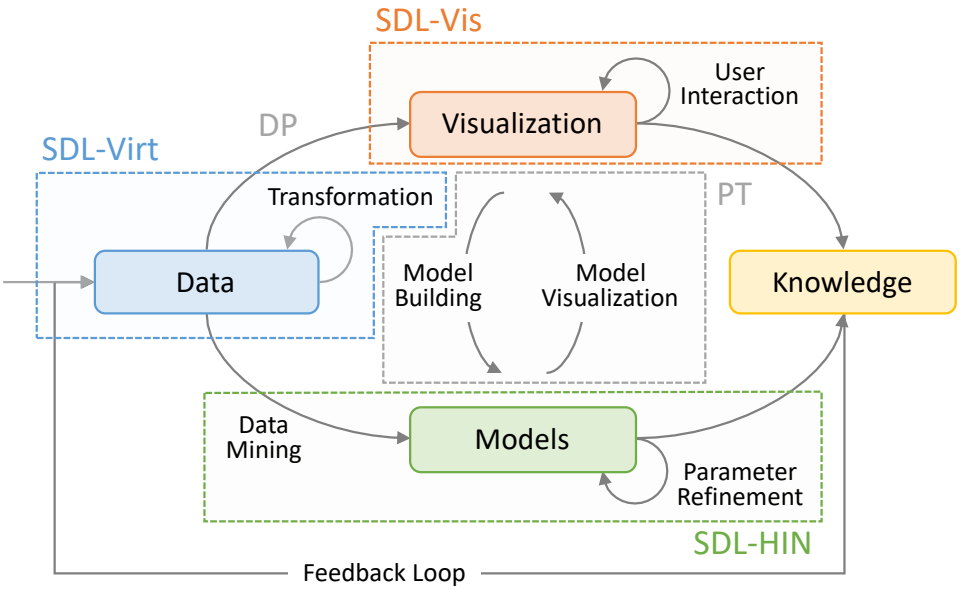

Figure 12: Integration of all SmartDataLake components into the Visual Analytics process. The tasks Parameter Tuning (PT) and Data Profiling (DP) connect the SDL-Vis layer with SDL-Virt and SDL-HIN.

3.3.1. Visual Analytics Engine

3.3.1.1. Short description

The Visual Analytics Engine communicates with SDL-Virt and SDL-HIN, acting as a backend interface between those components and the Visual Explorer. It therefore abstracts access to the lower levels of SDL and keeps user interfaces separated from analytical computations. To enable reusability and allow external access, the functionality of the Visual Analytics Engine is exposed in the form of a Visualization API. The Visual Analytics Engine implements the theoretical *visual analytics model*, as well as the *vertical optimizations* functionality.

3.3.1.2. Role in SDL

The Visual Analytics Engine is the intermediate layer between the visualization components and the lower-level layers of the SDL software stack. It prepares the data from SDL-Virt and SDL-HIN for visual inspection, while simultaneously offering abstract mechanisms for vertical optimizations. By exposing its functionality via an API, it decouples the components and enables reusability and extendibility.

3.3.1.3. Main functionalities

The following list outlines the planned functionalities for this component:

- *Visual analytics model*: A model for the orchestration of all components relevant for the knowledge generation in large data lakes, involving data access, mining algorithms and interactive visualizations. It defines a framework structuring and defining the interplay of these components, building the theoretical baseline for the visual analytics engine.
- *Vertical optimizations*: This includes optimizations that cross-cut the three layers, i.e., the visual analytics, the HIN mining, and the data virtualization engine, in order to further increase the scalability and interactivity of the provided visualizations. These optimizations focus on two aspects. The first involves introducing data access sessions, which will enable more efficient communication between the three layers, e.g., enabling the data virtualization engine to predict the access patterns and better organize the data accesses. The second involves pushing data-intensive analytics and mining algorithms down to the data virtualization engine, which will enable construction of better query plans and open opportunities for optimization, both in terms of IO and processing costs.

3.3.1.4. Architecture

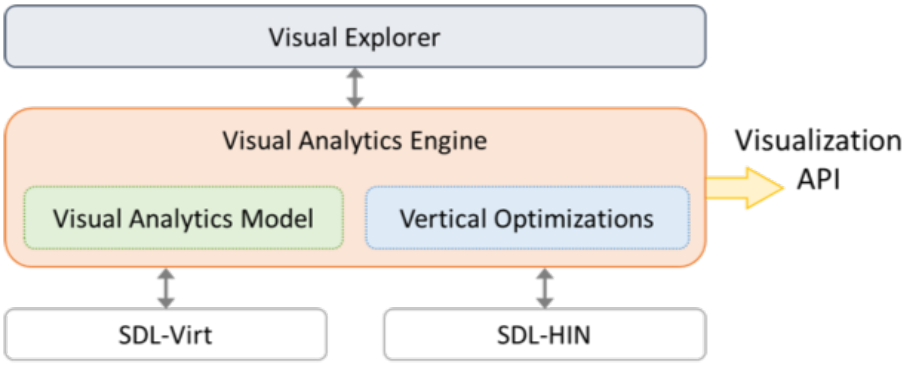

Figure 13: Architecture of the Visual Analytics Engine, connecting the Visual Explorer with SDL-Virt and SDL-HIN. Its functionality is exposed as Visualization API.

Figure 13 illustrates the main components of the Visual Analytics Engine. It interfaces SDL-Virt and SDL-HIN with the Visual Explorer, abstracting the access to the lower levels by a visualization API. The Visual Analytics Engine implements and therefore instantiates the theoretical visual analytics model as well as the vertical optimizations functionality.

3.3.2. Visual Explorer

3.3.2.1. Short description

The main goal of the Visual Explorer is to involve the user into the automated analysis by means of scalable, flexible, and interactive visualization techniques. A typical workflow involves:

- inspecting the underlying data assets to obtain some initial insights or formulate hypotheses;

- deciding on certain operations to be performed and selecting initial values for the involved parameters;
- inspecting the results to determine the next steps, which may involve, e.g., selecting different operations to be executed, changing the values of the parameters and re-executing the previous operations, including different or additional data sources in the analysis.

In particular, the two main aspects that this involves are presented below.

Feature exploration and parameter tuning. A data lake includes vast amounts of heterogeneous data from diverse sources. Hence, data profiling is an important part of the process in order to assist and guide the data scientist in formulating initial hypotheses about the data, steering the feature extraction process and determining which operations to perform and how. This provides guidance for determining, for instance, which attributes to use when searching for similar or near-duplicate entity profiles or grouping related entities together to identify communities in the network.

Scalable and interactive visualizations. To better assist the data scientist, the results of the various operations performed by the system should be presented by means of comprehensive visualizations. Appropriate visualizations for the various types of results will be supported. This ranges from basic types of charts for visualizing descriptive analytics over the data to tailored visualizations for spatial, temporal and graph data. By providing appropriate views and interaction possibilities, the data scientist is enabled and guided in his task of exploring the feature space as well as selecting and tuning parameters and models for the analysis.

3.3.2.2. Role in SDL

The role of the Visual Explorer is to visually support the user in using the tools and mechanisms of SDL in analyzing the contents of a data lake and extracting knowledge from it. Given that a data lake contains large amounts of information from multiple, heterogeneous sources, this is a largely exploratory process. The Visual Explorer offers visual assistance to the user throughout these tasks.

The Visual Explorer interfaces both with SDL-Virt, to allow inspecting the underlying contents of the data lake, and with SDL-HIN, to allow for tuning the parameters of the provided functionalities and for visualizing the results to determine the subsequent courses of action. Thus, it acts as a multi-faceted view on data foundation, analysis results and decisions of mining algorithm.

3.3.2.3. Main functionalities

The following list outlines the planned functionalities for this component:

- *Data profiling*: This refers to computing and visualizing various descriptive analytics about the input data. For instance, given a relational table, this includes determining the type of values in each column (e.g., numeric, categorical, textual) and, accordingly, various statistics, such as mean and standard deviation (for numeric attributes), number of distinct values (for categorical attributes), number of missing values, etc. Similarly, given a HIN,

statistics on the various meta-paths will be computed and visualized in order to assist the user in selecting certain meta-paths to be used for link prediction and community detection.

- *Parameter tuning*. The operations provided by SDL-HIN involve various parameters. For example, retrieving pairs of similar entities requires determining which attributes to use for similarity computations as well as setting a similarity threshold above which two entities are considered to be similar. This component will provide visual support to the user for tuning such parameters. For instance, by visualizing appropriately selected, indicative results of similar entities, the user will be guided towards increasing or decreasing the similarity threshold or modifying the similarity conditions.
- *Results visualization*. Tailor-made visualizations will be provided for the different types of results. These will include different types of charts for descriptive analytics based on the type of data (e.g., numeric versus categorical), map-based visualizations for spatial data, timeline visualizations for time series, and graph visualizations for HINs. These visualizations will be scalable, which will be based on appropriately ranking and summarizing the input data to be visualized (e.g., selecting which points to show on a map or which graph nodes and edges to include in a certain view of the network). They will also be interactive, meaning that the user will be able to perform certain actions to request more information, e.g., by zooming in and out, or selecting a specific object and focusing on it for more details.

3.3.2.4. Architecture

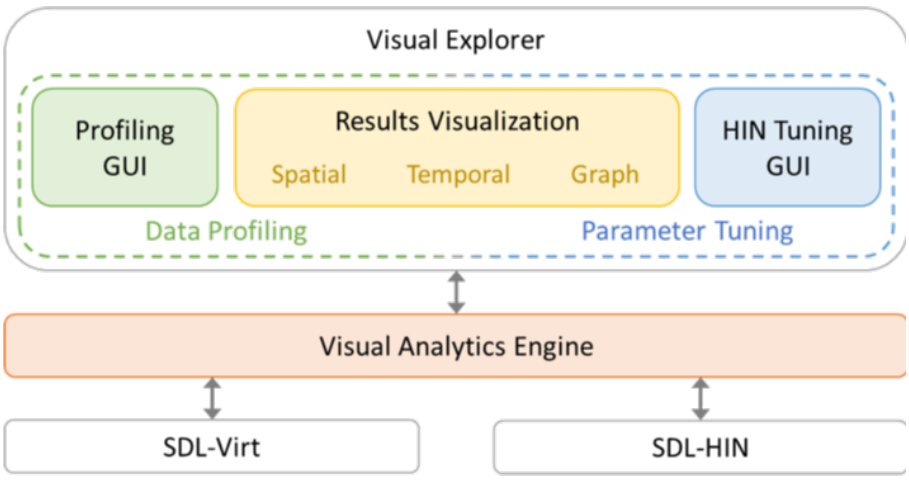

Figure 14: Main components of Visual Explorer.

Figure 14 illustrates the main components of the Visual Explorer, as described below:

- The task of inspecting and profiling available data assets in the data lake is referred to as *Data Profiling*. To enable this task, the visual analytics system provides specific user interfaces, allowing efficient exploration of the large amount of heterogenous raw data sources as well as derived information networks. The results are visualized to assist the

data scientist in obtaining initial insights about the data and formulating hypotheses to guide the next steps of the data exploration and analysis process.

- To configure and optimize the involved parameters of each executed operation (e.g., error bounds for approximate query processing, similarity thresholds for similarity joins, etc.), *Parameter Tuning* is essential. The visual analytics system supports this task by providing access to the inputs and parameters of the information mining engine, enabling user-steered optimization of such parameters. Once the operation is executed based on a distinct parameter set, the results are visualized and presented to the data scientist.
- The *Results Visualization* component provides a variety of visualizations of different types. This includes typical types of charts as well as more complex visualizations for spatial, temporal and graph data. By separating the *Profiling GUI* (Data Profiling) and the *HIN Tuning GUI* (Parameter Tuning) from the visualization components, an easy adaption of the views to specific tasks is possible.

4. SDL Reference Architecture

The SDL Reference Architecture comprises a number of additional components assumed to be available in an existing production setting or deployed to accommodate the needs of SDL's operation. These are not parts of the SDL platform as presented in Section 2, as they are instantiated according to operational requirements, but are considered as parts of the Reference Architecture of SDL, i.e., a broader collection of software artifacts and computing infrastructures available in a production setting. Indicatively, these include underlying data engines and file collections, data asset catalogue, external ETL pipelines, resource management system, orchestration, interactive CLI/web clients for exploratory data analysis, etc.

The Reference Architecture (i.e., SDL platform and external components) will be instantiated, deployed and tested in our integration and testing facilities, serving as a comprehensive deployment of SDL. This will facilitate our integration, testing and experimental evaluation but also provide a realistic training, evaluation and demo platform to inform industrial stakeholders about the benefits of SDL. In the context of our pilots, our industrial partners will adapt this Reference Architecture according to their individual operational requirements, existing software and computing infrastructures, as well as analysis workflows, instantiating select components of the SDL platform and integrating it with any existing systems and business processes in place.

In the following, we present the SDL Reference Architecture and its components.

4.1. Overview

The SDL Reference Architecture comprises the SDL Platform components identified in Section 2, in addition to several software modules, systems, libraries, and computing infrastructures briefly enumerated below and elaborated in Section 4.4.

In Figure 15, we present all components and relationships of the SDL Reference Architecture. Specifically:

- **Data Management Infrastructure**. This part encapsulates all possible sources of data assets identified by the system owner to be accessible by the SDL data lake, i.e., CSV files, relational database management systems, NoSQL data management systems, XML/JSON stores. These are external data sources to the SDL platform and under the full control of the system owner, which defines access-level controls to the data as desired.
- **Computing Infrastructure**. This part encapsulates the cloud infrastructure upon which all components of the SDL Reference Architecture are deployed and executed. Hence, it serves two purposes: (a) host system services assumed to be constantly available for the system to operate (e.g., the AAI component, a front-facing client IDE), and (b) provide the required computing resources for the scalable computation of SDL analysis algorithms within the established target environment (e.g., execution of an entity resolution algorithm). It provides the cloud storage and IaaS services for the containerized deployment and scaling (up/down, in/out) of software components, and accommodates the massively scalable execution of computing jobs via distributed and parallel execution.
- **Resource Management**. This part is responsible for the management, orchestration and invocation of all data analysis workflows and the allocation of computing resources in accordance to demand and requested level of service. It comprises a containerization management system for initializing and deploying system-wide services with highly granular and specialized control over the required target environment and scaling policies (e.g., an algorithm requiring a few VMs with high memory vs. an algorithm requiring a high number of VMs with multiple virtual cores). Moreover, it performs the management and execution of workflows, including both system-level tasks, as well as analysis jobs (e.g., a DAG of user-defined HIN analytics and visualization algorithms). As such, it allocates computing resources, deploys target environments, initializes the appropriate SDL components, executes all workflow steps, manages any interim results, and delivers the output.
- **Access Control**. This part implements a platform-wide secure authentication and authorization infrastructure for users and user roles, API endpoints, data assets, and resources. It allows the discovery and use of all services provided by SDL and their invocation (atomic or in workflows) in a highly granular manner, ensuring the isolation of assets, services, and resources from different users and organizations sharing the same platform.
- **Clients**. This part includes the core client services enabling the direct use and integration of SDL in existing third-party systems and workflows. A CLI supports the programmatic use of SDL services from users and systems, ensuring wide compatibility with typical data analysis workflows. A visual IDE abstracts the underlying complexity, enabling non-experts to visually design analysis workflows using SDL platform's services and assets. Finally, Jupyter notebooks support interactive computing scenarios, directly focusing on data scientists aiming to harness the core benefits of SDL.

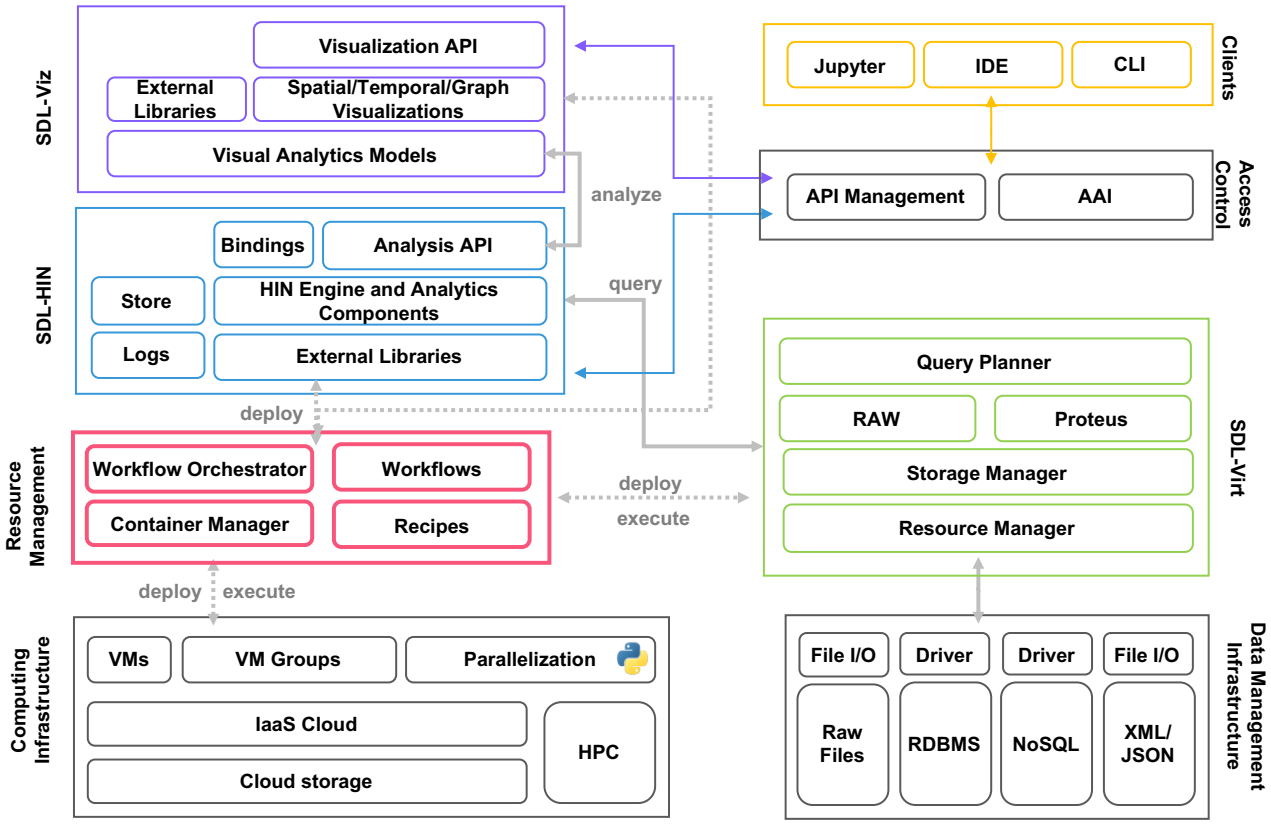

Figure 15: SDL Reference Architecture.

4.2. Usage Contexts

Before continuing with the presentation of the SDL Reference Architecture components, and to assist the reader in understanding the operation of the platform in real-world settings, we consider two end-to-end usage contexts detailing the internal data and process flows.

4.2.1. Context I: Interactive computing

In this context, SDL is used in a typical interactive computing scenario by a data scientist via, for instance, the Jupyter notebook Web interface and in a programming language of her choosing (e.g., Python).

A Jupyter notebook[3] is considered as a collection of atomic operations executed in sequence or ad hoc within the same scope. An operation typically receives as input the output of another operation. The execution of each operation is handled by the underlying Jupyter notebook server within its established context (i.e., libraries, computing environments/resources). For our purposes, operations can be categorized as:

- Context-specific (e.g., authenticate user, import data file). These are provided both by Jupyter (e.g., file import) and the SDL platform (e.g., define the user's context) as needed.

[3] https://jupyter.org/

- Data access (i.e., query data). These are provided both by the SDL platform, externalizing access to the underlying data lake, as well as database specific adapters and backends outside SDL.
- Native data processing (i.e., custom data processing and/or visualization via native programming language constructs). These contain all operations expressed using the standard facilities of the selected programming language. The execution is handled by the Jupyter notebook server within its specified target environment.
- External libraries (i.e., data processing and/or visualization via external libraries available in the notebook context). These contain operators available from external processing libraries. The execution is performed according to the capabilities of each library (e.g., parallel and distributed processing over Apache Spark).
- SDL analyses (i.e., data processing and/or visualization via components of the SDL platform). These are operators provided exclusively by SDL-Virt, SDL-HIN and SDL-Vis modules. They are executed according to their individual implementation characteristics over the platform's computing infrastructure.

Within this setting, a typical user workload comprises one or more of the following operations.

- *Set user context*. The user enters her unique API key as provided by the platform, which establishes the broader context for authorization to data assets, analysis algorithms, and type/intensiveness of computing resources. This ensures that the platform's policies and SLA's per user/organization are uniformly enforced, while it also enables appropriate adjustment of user privileges.
- *Discover and invoke available data assets*. The user, based on the access rights established in her context, has access to a catalogue of available data assets (including appropriate metadata) from the underlying data lake, which she can query and visualize via the corresponding methods exposing internal system APIs in the programmatic framework set (e.g., Python). The user does not have direct access to internal platform services via other means, nor can she manipulate the data lake (e.g., add or remove data sources) as this is performed by the data lake administrators. Query output can be further processed and analyzed as-is (via native programmatic language operators), or via exposing it in an appropriate construct.
- *Discover and invoke analysis and visualization algorithms*. The user, based on the access rights established in her context, has access to a catalogue of available analysis services (as well as instantiated workflows of analysis services), which she can discover, invoke and configure via the corresponding methods exposing the available analysis algorithms in the programmatic framework set. The execution of the analysis and visualization algorithms is performed outside the scope of the notebook and in an appropriate computing framework and infrastructure as established for each individual algorithm (e.g., parallel and distributed processing over Apache Spark) and user context (i.e., resource utilization quota and SLA). All steps of the execution are automatically handled by the SDL platform, ensuring support for all types of processing requirements, as well as optimal utilization of resources. Analysis output is returned and can be further used as-is (via native programmatic language

operators), or via exposing it in appropriate constructs. Finally, visualization output is returned and embedded in the notebook interface, with any interactive UI components (e.g., slider for timelines, panning for maps) included in the visualization.

- *Process data natively*. The user can invoke any operator (native programmatic framework construct or external library) to process and analyze the available data assets as required. Hence, the analysis and visualization algorithms of SDL can support broader ad hoc exploratory and analysis workflows and can be integrated in existing notebooks used by the involved data scientists to augment them with specialized and efficient analysis and visualization operations.
- *Operationalization*. A subset of the notebook's processing and analysis operations can be selected and exported by the user as a standalone and reusable analysis component to be programmatically invoked by external data processing workflows and systems. The component is persisted as a new analysis workflow by the SDL platform within the user's context (i.e., authorization to data assets, analysis algorithms, and type/intensiveness of computing resources) or outside its context (e.g., with additional computing resources allowed) and becomes available for integration from third-party systems by the SDL's APIs and supported bindings to programmatic frameworks.

4.2.2. Context II: Scalable computing

In this setting, the system is being used in a typical scalable computing scenario by invoking an SDL analysis algorithm or a persisted analysis workflow as a standalone component or integrated within a complex data analysis workflow by an expert or a third-party system. The explicit assumptions are that (a) invocation of SDL analysis algorithms and workflows result into long execution times (e.g., hours/days), and (b) the configurations of the SDL algorithms and workflows, as well as any pre-processing required for preparing their respective input has been crystallized by an expert user (e.g., following an interactive computing exercise and/or by applying domain knowledge).

A typical user workload is much simpler when compared to the one presented in Section 4.2.1, as it only involves commands for (a) setting the user's context, and (b) invoking an SDL analysis component or persisted workflow.

4.3. Deployment Options

SDL by design follows an adaptive and extensible architecture allowing for the real-world deployment of select parts of its software stack, due to the operational requirements established in *Deliverable D1.1 "Use Cases and Requirements"*. As such, an industrial stakeholder with a data lake already in place, should be able to harness the analytics and visualization components of SDL by integrating them in its infrastructure. In another scenario, complex established data processing and analyses workflows can exploit individual SDL components by invoking the corresponding software as services (SaaS).

In the following, we briefly present and discuss the motivating usage paradigms covering the real-world deployment and use of the SDL platform. Our presentation is by no means exhaustive and can be extended in the future in case of novel emerging use cases and paradigms. However, it serves the purpose of framing our ambition and demonstrating the flexible support of SDL for a plethora of real-world industrial applications and settings.

4.3.1. Deployment of SDL-Virt

The owner opts to deploy only SDL-Virt without other SDL platform modules. In this case, the data API offered by SDL-Virt is exploited to provide virtualized, homogeneous access to the underlying data assets to all existing external data analysis, mining, and visualization software components owned by the user. As such, SDL only serves to efficiently support data discovery and querying. This setting implies:

- *ETL*. Any existing relevant ETL pipelines need to be replicated and/or migrated to SDL's data lake ensuring full expressive compatibility.
- *Data access*. SDL's data lake must provide discovery services and querying access to its data assets through syntactically and semantically homogeneous patterns facilitating the migration of any existing analysis pipelines.
- *Workloads*. The workloads expected to be supported concern analytical and interactive knowledge discovery queries with bindings over multiple programming frameworks.

The deployment of the SDL platform in this case refers to its use as an off-the-shelf data lake engine in tandem with multiple other data/workload-optimized database engines and repositories.

4.3.2. Deployment of SDL-HIN

The owner deploys only the components of SDL-HIN to perform large scale analysis of HIN data assets as atomic components or broader persisted workflows. Data exploration, feature engineering, and parameter tuning are performed by processes and systems outside the scope of SDL. SDL only provides the building blocks for scalable analysis over HINs in a workload- and visualization-agnostic manner. This setting implies:

- *Data provision*. Data is either provided as HINs (i.e., a readily available network of entities) or must be transformed to a HIN according to case-specific modeling requirements. The HIN analytics components are agnostic in terms of the software/system used to prepare and provide the data assets.
- *Integration*. Analytics components are self-contained artifacts responsible for efficiently scaling according to the target computing resources. The provided execution logs, scalability patterns and target workloads serve to inform the owner regarding the optimal resource allocation.
- *Workflows*. The analysis components are invoked as atomic entities, within HIN analysis workflows, or within broader analysis workflows involving external libraries. The invocation and orchestration of components outside the SDL-HIN layer is handled by the owners.

- *Visualization-agnostic*. A visualization component to manipulate, preview or in any way guide the analysis process is not assumed to exist nor provide any specialized functionalities.

The deployment of SDL platform in this case serves the ad hoc and periodic programmatic definition, orchestration, and invocation of scalable analytics with minimal interactivity requirements.

4.3.3. Deployment of SDL-Vis

The owner deploys the components of SDL-Vis to support interactive visual analytics over existing data assets and/or analysis results offered by external services. This setting implies:

- *Web notebooks*. The primary interface for interactive computing is the Jupyter notebook, a de facto operational standard for data scientists with diverse support for a plethora of programming languages, frameworks, and libraries. SDL's analysis and visualization artifacts must provide support for notebooks and language bindings in an extensible manner.
- *Computing infrastructure*. The execution of notebooks is provided by generic-purpose (i.e., a single server) or specialized computing infrastructures (e.g., IaaS cloud, HPC) provided by commercial vendors (e.g., Amazon) or the owner. The execution of SDL's components must be possible on low-cost and easily affordable infrastructures (i.e., local workstation), while also taking advantage of more complex services to increase performance, interactivity and cost-effectiveness.
- *Data sources*. SDL's components need to support the manipulation and analysis of data assets in simple formats (i.e., files) without assuming their efficient provision from a data engine. As such, the components need to ensure scalability and interactivity in a self-sustained manner.
- *Streamlined operationalization*. The final output of data analysis and knowledge extraction by data scientists must be available for streamlined and timely operationalization in a production setting. Persisted data analysis workflows are considered as decoupled from ongoing improvements aiming to increase effectiveness, accuracy, insights, and business relevance.

The deployment of SDL platform in this case serves knowledge extraction by data scientists in accordance to the interactive computing paradigm.

4.3.4. Deployment of full SDL stack

In this case, we assume that the complete SDL platform is deployed, initialized, and enters production by an industrial stakeholder. All components of the SDL Platform (as presented in Section 2) are available and used. This setting implies:

- *Reference architecture components*. All components identified in the SDL Reference Architecture are instantiated and are available to the SDL platform. The owner is

responsible for providing or procuring (e.g., commercial IaaS cloud) all required computing resources, handling their allocation according to operational requirements.

- *Data Lake*. The owner has no data lake system in place or opts to replace an existing system with SDL's data lake due to its performance, scalability, and efficiency benefits. SDL supports the initialization and operation of the data lake with all required data assets.
- *Pipelines*. The owner has no existing data processing/analysis/visualization pipelines in place or opts to replace them with pipelines provided by SDL. Any existing data processing pipelines need to be seamlessly replicated by, or migrated to, SDL.
- *Workloads*. The system is required to support both interactive and scalable analysis workloads, i.e., serve data scientists and data analysts/DSS applications.

The deployment of the SDL platform in our testing environment (see Section 5.3) implements this case, in which analysis and visualization facilities serve both interactive and scalable analysis requirements.

4.4. Components

In the following, we present the components of the SDL Reference Architecture that are not part of the SDL platform. For each component, we provide a short description of its role in the context of the SDL Reference Architecture, and our current approach for its instantiation. The presentation that follows is by no means exhaustive or final, as the software components realizing our Reference Architecture and their operation may change to accommodate the project's needs.

4.4.1. Data Management Infrastructure

This component encapsulates all possible sources of data assets identified by the system owner to be accessible by the SDL data lake, i.e., CSV files, relational database management systems, NoSQL data management systems, XML/JSON stores, graph stores. These are external data sources to the SDL platform and under the full control of the system owner, which defines access-level controls to the data as desired.

During the implementation of the Reference Architecture, we will setup several data stores for managing HIN data assets. Initially, we plan to install and configure a Network File System (NFS) service, the Hadoop Distributed File System (HDFS[4]) and a Neo4j[5] native graph database instance. The list of the data stores is not exhaustive and during the project, depending on partner feedback, more data stores may be installed.

4.4.2. Computing Infrastructure

This component encapsulates the cloud infrastructure upon which all components of the SDL Reference Architecture (e.g., services, algorithms) are deployed and executed. Hence, it serves two purposes: (a) host system services assumed to be constantly available for the system to operate

[4] https://hadoop.apache.org/

[5] https://neo4j.com/

(e.g., the AAI component, a front-facing client IDE), and (b) provide the required computing resources for the scalable computation of SDL analyses algorithms within the established target environment (e.g., execution of the entity resolution algorithm). It provides the cloud storage and IaaS services for the containerized deployment and scaling (up/down, in/out) of software components, and accommodates the massively scalable execution of computing jobs via distributed and parallel execution.

At the hardware level, several virtual machines will be instantiated for hosting the required software. On top of the hardware virtualization lever, containers will be also instantiated and managed using either the Docker Swarm [6] platform or the Kubernetes [7] system. A detailed description of the cloud infrastructure used for virtualization is available in Section 5.3.

4.4.3. Resource Management

This group contains the components that are responsible for the management, orchestration, and invocation of all data analysis workflows and the allocation of computing resources in accordance to demand and requested level of service. It comprises a containerization management system for initializing and deploying system-wide services with highly granular and specialized control over the required target environment and scaling policies. It also performs the management and execution of workflows, including both system-level tasks as well as analysis jobs. As such, it allocates computing resources, deploys target environments, initializes the appropriate SDL components, executes all workflow steps, manages any interim results, and delivers the output.

4.4.3.1. Container Manager

The Container Manager is responsible for creating containers for either hosting the SDL services and applications or executing analysis algorithms. In SDL, we will be using either the Docker Swarm platform or the Kubernetes system as the Container Manager. Containers hosting SDL services and applications will be long-lived containers and will be created during the initial system deployment. In contrast, containers used for algorithm execution will be created on demand by the Workflow Orchestrator component or the computational framework used (e.g., Apache Spark[8] or Dask[9]).

4.4.3.2. Workflows

A Workflow consists of one or more tasks that are organized as a directed acyclic graph (DAG). It enables the modeling, definition, and invocation of complex analysis pipelines comprising an arbitrary number of atomic operations which can be executed sequentially, in parallel, or conditionally. A task may perform a simple job such as copying files between two hosts or a complex computation, such as any of the HIN analysis algorithms provided by SDL-HIN. A workflow can be ephemeral (i.e., serve a specific analysis task) or persisted (i.e., leveraged as a new

[6] https://docs.docker.com/engine/swarm/

[7] https://kubernetes.io/

[8] https://spark.apache.org/

[9] https://dask.org/

atomic operation). Workflow execution is managed by the Workflow Orchestrator component which is responsible for initializing and executing workflow instances. Simple tasks may be executed in process. Nevertheless, more complex tasks will be executed in external containers provided by the Container Manager component.

4.4.3.3. Recipes

In some cases, each Workflow instance may be executed once. Nevertheless, sometimes Workflow instances may have to execute repeatedly due to changes to either the underlying data or the task configuration parameters. A *recipe* is a special instance of a Workflow which is partially configured and can be used as a *template* for creating a new workflow instance, without having to update all the configuration parameters. Recipes will be managed by the IDE client descripted in Section 4.4.5.2.

4.4.3.4. Workflow Orchestrator

The Workflow Orchestrator is responsible for the initialization and execution of workflow instances. For each task, it prepares the required input and configuration files, requests containers from the Container Manager, monitors the execution of the task and collects any generated output files including logs, KPIs and sample data. Moreover, it handles execution failures and, optionally, restarts failed tasks if possible.

The Workflow Orchestrator also creates detailed log entries for each workflow and task execution and makes this information available to the IDE client.

4.4.4. Access Control

This component implements a platform-wide secure authentication and authorization infrastructure for users/user roles, API endpoints, data assets, and resources. It allows the discovery and use of all services provided by SDL and their invocation (atomic or in workflows) in a highly granular manner, ensuring the isolation of assets, services, and resources from different users and organizations sharing the same platform.

4.4.4.1. API Management

The CLI client implements a RESTful API for accessing SDL-Virt, SDL-HIN and SDL-Vis functionalities. The authentication and authorization control of the API endpoints is handled by using application keys. Application keys are generated by the IDE client application and are assigned to an existing user account. Every API invocation is executed in the security context of the assigned user. For each API call, the API Management component logs additional information in addition to the log entries created by the Workflow Orchestrator. Moreover, API Management enforces constraints on the number of concurrent and total requests per application key as well as at the system level to implement resource throttling.

4.4.4.2. AAI

The goal of the Authentication and Authorization Infrastructure (AAI) component is to provide a common method for sharing a security context among the several components of the SDL platform. In the SDL platform, we plan to use the Keycloak[10] service for interacting with the existing authentication infrastructure used by each project partner. The authenticated users will be mapped to SDL users managed by the IDE client. Hence, the IDE client will support user whitelisting and fine-grained access control to the SDL platform components.

4.4.5. Clients

This group includes the core client services enabling the direct use and integration of SDL in existing third-party systems and workflows. A CLI supports the programmatic use of SDL services from users and systems, ensuring wide compatibility with typical data analysis workflows. A visual IDE abstracts the underlying complexity, enabling non-experts to visually design analysis workflows using SDL platform's services and assets. Finally, Jupyter supports interactive computing scenarios.

4.4.5.1. CLI

CLI consists of all programmatic libraries and APIs used for accessing the SDL services. We plan to develop the following software artifacts as part of the CLI component:

- A RESTful API for accessing SDL-Virt, SDL-HIN and SDL-Vis functionality over a secure HTTP connection. The API endpoints will have support for either OAuth 2.0 authentication or application keys.
- Python or Java bindings for the API for integrating the API into existing applications.
- Python modules aimed for use inside Jupyter notebooks.

4.4.5.2. IDE

The purpose of the IDE is to abstract the underlying complexity and allow users to easily perform the following tasks:

- Query a catalogue of HIN data asset metadata and discover data sources. The catalogue is populated either with existing datasets or data generated by workflow execution instances that publish their output to the catalogue.
- Invoke single analysis and visualization algorithms using predefined configuration files and data assets available in the catalogue.
- Visually design complex workflows and schedule their execution on the underlying computing infrastructure. Optionally, visualize the results of a workflow execution instance directly within the IDE user interface.
- Manage SDL platform users and permissions.

[10] https://www.keycloak.org/

- Manage application keys for accessing the RESTful API.

4.4.5.3. Jupyter

The Jupyter notebook is the de facto operational standard for data scientists with diverse support for a plethora of programming languages, frameworks, and libraries. In SDL, we are going to configure an appropriate Jupyter Notebook Server for securely accessing the SDL Reference Architecture underlying computing infrastructure and provide libraries for submitting jobs to the installed computational frameworks such as Apache Spark or Dask. The specific server will be available only to authorized users. In addition to using a custom server, users will have the option to use Python modules that will wrap the RESTful API, developed as part of the CLI component, and execute analysis or visualization algorithms.

5. Integration and Deployment

In this section, we present the software development principles adopted in the project and the methodologies applied for integrating and testing all developed software.

5.1. Software Development Principles

In the following, we provide a brief overview of the principles and methodologies followed in the project for software development, integration, testing, and release management. In summary, we employ an Agile approach and the RERO (Release Early, Release Often) paradigm. Our methodological framework has been shaped both from our previous experiences in software development for open source projects (e.g., GeoKnow[11], DAIAD[12], SLIPO[13]), as well as the requirements of SDL in terms of complexity, effort, and time constraints.

5.1.1. Agile

Next, we highlight specific Agile[14] practices employed in the project and, when required, the appropriate context for each one.

- *Pair-programming*. Software developers and researchers (PhD candidates, Research Associates) spend 10-30% of their time (depending on the actual task/functionality and importance) in pair-programming. This approach is also applied during internal testing (unit/integration) to speed-up the discovery and redress of bugs and issues.

[11] https://github.com/GeoKnow

[12] https://github.com/DAIAD

[13] https://github.com/slipo-eu

[14] http://www.agilemanifesto.org/

- *Software that works*. Our emphasis is placed on delivering working software, deployed and tested on a production setting. This is sound both from a methodological perspective but also needed to address the complexity of the complete SDL platform.
- *Continuous change and development*. New or changed user requirements will be continuously collected from stakeholders and transferred in the development cycle, ensuring quick response to change.

5.1.2. Release Early, Release Often (RERO)

The RERO paradigm has its foundations deeply rooted in open source projects[15] (e.g., Linux kernel), focusing on maintaining momentum in development, as well as accelerating feedback received from end-users and testers. Its opposite is a strict and feature-based release schedule (i.e., release only if a new functionality is complete). Once again, the characteristics of the SDL system and its development timeframe favors rapid vs. infrequent feedback (e.g., weekly vs. quarterly). The implementation of RERO in the project will focus on short (1-3 weeks) development and testing cycles (i.e., sprints) across all various system components (e.g., APIs, libraries, UI elements).

5.1.3. Benevolent dictatorship and tight commit control

The governance and commit/attribution models in open source projects[16, 17] vary greatly depending on their foundations, age, and popularity. The individual open source software components comprising the SDL platform (see Section 3) will continue to follow their already established governance and commit control guidelines.

The remaining SDL components developed for the needs of the project consist a new open source project, focused on an innovation-centric agenda, with zero external contributions, as well as a very tight timeframe for developing, testing, and validating its output. Towards this, we will adopt the following governance and collaboration directions.

- *Benevolent dictator*. Software development is steered by a single person with exclusive control over the directions and course of the software. With no external contributors (i.e., outside the project partners), this is the preferred governance model for young open source projects (especially during their incubation phase).
- *Commit Control*. The roles of code authors vs. committers are typically used interchangeably in open source development, but they are not actually the same. Commit roles/rights are a matter of governance (commit access), policy (public attribution as incentives for participation) and technology (centralized vs distributed repos). In SDL, a Software Lead will be assigned to each software component in the project's GitHub[18]

[15] http://firstmonday.org/article/view/578/499

[16] http://oss-watch.ac.uk/resources/governancemodels

[17] http://producingoss.com/

[18] https://github.com/smartdatalake

organization repository. Code authors will submit contributions for each project to the assigned Software Lead. Each lead is responsible for receiving, reviewing, improving, or requesting further improvements from the authors. If the Leads are satisfied, the source code is committed by them in the corresponding public repository. This approach is proposed due to the strict timeframe of the project for delivering its beta prototype, the high number of developers involved, and the diverse software developed.

5.2. Integration and Testing

In the following, we summarize our methodology and practices for integrating and testing all software developed in the project. The integration and testing methodologies are as follows.

- *Continuous integration* (CI). Source code contributions are integrated and tested at very frequent intervals (e.g., daily). Due to the API-based and loosely coupled nature of the SDL platform, CI is often performed interdependently across its various components. For example, CI for the SDL analysis algorithms can be performed as often as needed, since each one is a standalone application and communication with the SDL services is decoupled using wrapper components which can be easily replaced by testing mock components.
- *Integration testing*. We have adopted the following two methodologies for integration testing to accelerate development and ensure scalability.
    - Big Bang. This testing process is applied for the entire SDL platform, following specific usage workloads that cover all usage aspects of the system, testing infrastructure, tester groups, and data. The workloads used for testing will be defined during the project development.
    - Risky-hardest. System APIs, which by definition are slowly changing and affect the entire system operation, are additionally tested first using the Risky-hardest methodology. When a new API version is available that provides a solution to a previously identified problem or new functionality, integration testing begins by focusing on the API itself.
- *Staging and Production testing*. For development, integration, and testing purposes we will deploy several computing infrastructures presented in detail in the next section.

5.3. Deployment

For testing the SDL platform, we will deploy all of its components and instantiate the complete Reference Architecture within the Hellenic Data Service[19] (HELIX), the national cloud infrastructure for data intensive research and innovation co-developed by Athena RC, harnessing its highly scalable compute, network and storage resources. SDL is one of the multiple research and

[19] http://www.hellenicdataservice.gr

innovation projects hosted by HELIX for Big Data, ensuring its sustainable operation and implementation of our research and exploitation plans.

HELIX comprises three components offering services to scientists and researchers: HELIX Pubs[20], providing search facilities for Open Access publications authored by Greek scientists and researchers, acting as a national OpenAIRE[21] node and harvesting national institutional and thematic scientific repositories; HELIX Data[22], a data catalogue and repository for sharing, discovering, assessing, visualizing and downloading scientific data assets; HELIX Lab[23], an open-ended collection of services enabling the interactive and scalable manipulation, experimentation and analysis of Big Data assets. SDL will be deployed as a service of HELIX Lab (see Section 5.3.1), leveraging its underlying IaaS and HPC infrastructure for scalable data management, analysis, and visualization, as well as harnessing its existing software components (e.g., resource management, AAI) to instantiate the complete SDL Reference Architecture.

On a system level, SDL will be deployed on top of the Synnefo cloud stack[24]. Synnefo is a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services, similar to those offered by AWS[25]. On a hardware level, the production system will be hosted on Athena RC's private IaaS available from okeanos-knossos.gr, currently allocating 512 CPU cores, 512 GB main memory, 40TB storage, which can scale horizontally (scale-out) and vertically (scale-up) depending on utilization. Figure 16 presents the data center where the physical hardware is hosted.

Next, the Synnefo cloud stack, the HELIX Lab services, the SDL deployment scheme, and the deployment orchestration method are described.

Figure 16: SDL production servers hosted at the data center of the Greek Ministry of Education.

[20] https://pubs.hellenicdataservice.gr

[21] https://www.openaire.eu/

[22] https://data.hellenicdataservice.gr

[23] https://lab.hellenicdataservice.gr

[24] https://www.synnefo.org/

[25] https://aws.amazon.com/

5.3.1. Synnefo

Synnefo manages multiple Ganeti [26] clusters at the backend for handling of low-level VM operations and uses Archipelago[27] to unify cloud storage. To boost third-party compatibility, Synnefo exposes the OpenStack APIs[28] to users. The Cyclades (Synnefo UI) allows administrators to maintain and manage the cloud resources of the system (restart, shut down, start up, delete and create a virtual machine). In Figure 17, an overview of the Synnefo services is presented.

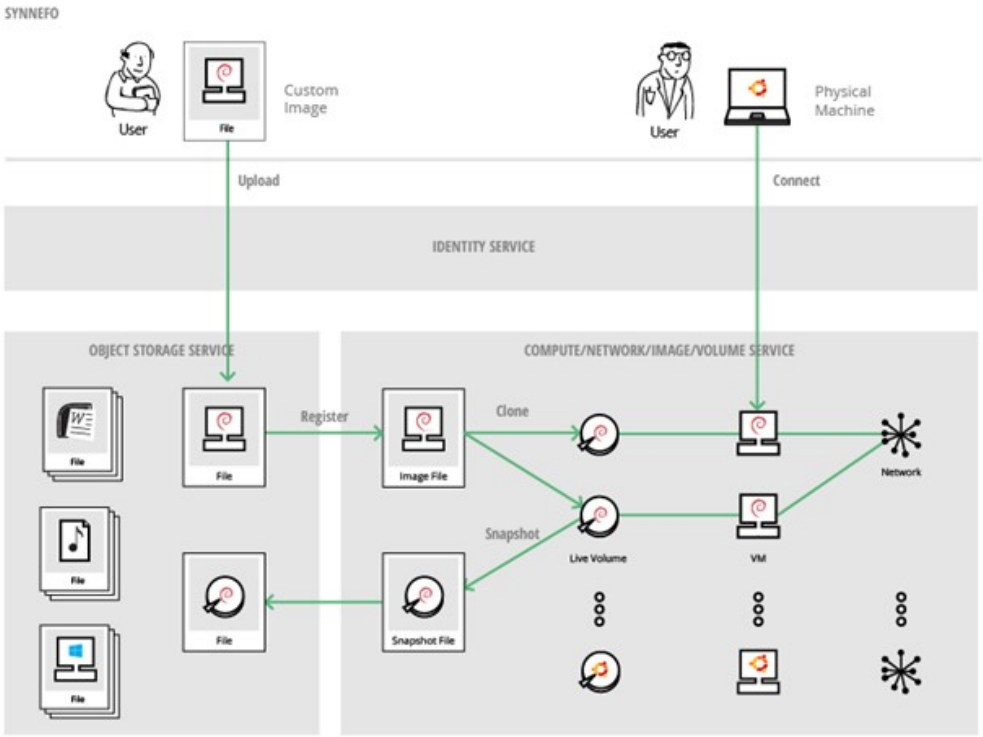

Figure 17: An overview of the Synnefo services.

Synnefo keeps a clear separation between the traditional cluster management layer and the cloud layer. This unique design approach leads to a completely layered architecture boosting production readiness, maintainability and upgradability. The modular design allows for linear scalability, gradual extensibility and ease of operations. In Figure 18, an overview of the Synnefo architecture is presented, showing the major components of each layer.

[26] http://www.ganeti.org/

[27] https://www.synnefo.org/docs/archipelago/latest/

[28] https://developer.openstack.org/api-guide/quick-start/index.html

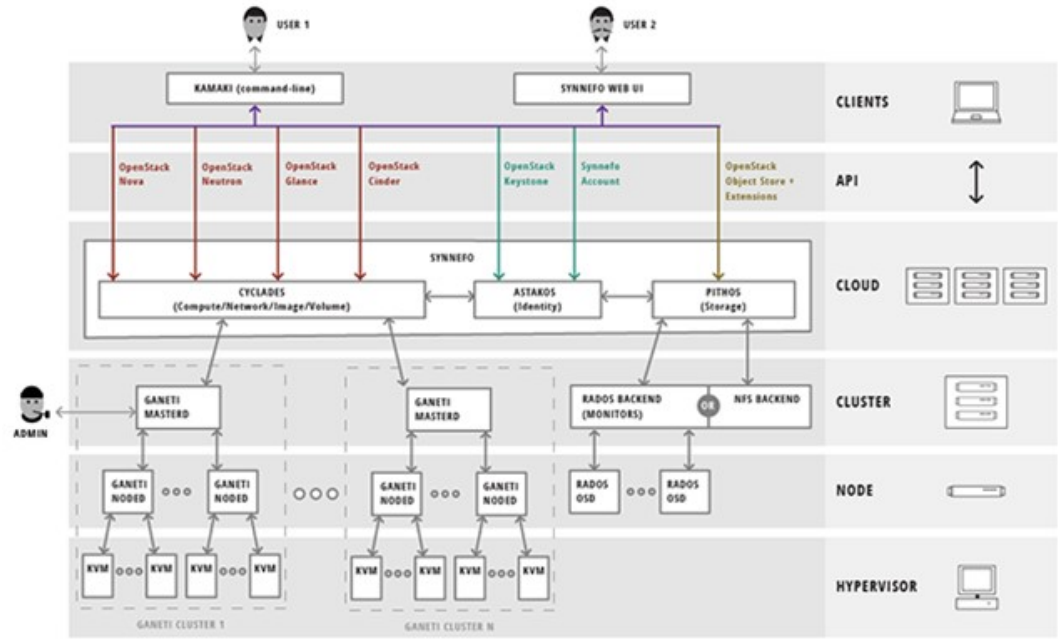

Figure 18: The system architecture of Synnefo

For all VMs inside the cluster, basic system-wide monitoring is available from the Synnefo layer. The collected statistics can be exported to a table-like format via Synnefo API, or (part of them) can be directly visualized from Synnefo UI.

5.3.2. Deployment Overview

The installation will comprise the following virtual machines (or machine groups):

- A single administration server that acts: (a) as a gateway for accessing the cluster private network and administrating hosts remotely using SSH, and (b) as a proxy for the HDFS, YARN and Spark administration sites.
- An application server (SDL webapp) hosting the Client IDE web application and exposing the RESTful API.
- An application server (SDL RPC service) receiving workflow-related requests from SDL webapp. The main responsibility of this component is to schedule, execute and monitor the status of workflows. The actual tasks of a workflow DAG are executed inside a dedicated Docker Swarm (Workflow Tasks).
- A JupyterHub server acting as a proxy to user-scoped Jupyter notebook servers.
- An Identity Provider (AAI) as the central point for user authentication. This service is a basic component of SDL deployment, but it need not live exclusively into SDL infrastructure (may be shared with HELIX infrastructure).
- A build server for managing software build/integration tasks. This also includes Ansible playbooks for managing installation/updates of applications hosted on other machines of the SDL deployment.

- A Hadoop cluster for hosting a distributed filesystem (HDFS) and for performing data-analysis tasks on it. The Spark computing engine will be deployed over Hadoop and will use YARN as its cluster/resource manager.
- A relational database (PostgreSQL) for the state of all web applications (SDL webapp and RPC service).
- A graph database.
- An NFS server for hosting data that need to be accessible by several machine groups (user directories, input files for workflows, periodic backups).

5.3.3. Deployment Orchestration

For facilitating the deployment of SDL, we will create a GitHub repository for storing the scripts and configuration files needed for the several node groups. The tool used to automate the deployment and update process is Ansible[29].

Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs. Being designed for multi-tier deployments since day one, Ansible models IT infrastructure by describing how all systems inter-relate, rather than just managing one system at a time. It uses no agents and no additional custom security infrastructure, so it's easy to deploy – and most importantly, it uses a very simple language (YAML, in the form of Ansible playbooks) that allows administrators to describe their automation jobs in a way that approaches plain English.

Ansible works by connecting to the system nodes and pushing out small programs, called "Ansible Modules" to them. These programs are compiled to (try to) achieve the desired state of the system (as described inside the playbook). Ansible then executes these modules (over SSH by default) and removes them when finished. The library of modules can reside on any machine, and there are no servers, daemons, or databases required. Typically, the administrator works with a terminal program, a text editor, and probably a version control system to keep track of changes (e.g., git[30]). Ansible reads a set of managed machines (organized into groups) from a simple INI/YAML configuration file. A playbook can orchestrate multiple slices of the infrastructure topology, with very detailed control over how many machines to tackle at a time.

The work of parameterizing and automating the deployment process will evolve as new software components of SDL are implemented.

[29] https://www.ansible.com/

[30] https://git-scm.com/

References

[Aggarwal2011] Charu C. Aggarwal, Yan Xie, Philip S. Yu: Towards Community Detection in Locally Heterogeneous Networks. SDM 2011: 391-402

[Boden2014] Brigitte Boden, Martin Ester, Thomas Seidl: Density-Based Subspace Clustering in Heterogeneous Networks. ECML/PKDD (1) 2014: 149-164

[Chen2017] Yuxin Chen, Chenguang Wang: HINE: Heterogeneous Information Network Embedding. DASFAA (1) 2017: 180-195

[Chen2018] Hongxu Chen, Hongzhi Yin, Weiqing Wang, Hao Wang, Quoc Viet Hung Nguyen, Xue Li: PME: Projected Metric Embedding on Heterogeneous Networks for Link Prediction. KDD 2018: 1177-1186

[Christophides2015] Vassilis Christophides, Vasilis Efthymiou, Kostas Stefanidis: Entity Resolution in the Web of Data. Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers 2015

[Chrysogelos2019] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, Anastasia Ailamaki: HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. PVLDB 12(5): 544-556 (2019)

[Dong2015] Xin Luna Dong, Divesh Srivastava: Big Data Integration. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2015, pp. 1-198

[Fier2018] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, Johann-Christoph Freytag: Set Similarity Joins on MapReduce: An Experimental Survey. PVLDB 11(10): 1110-1122 (2018)

[Getoor2012] Lise Getoor, Ashwin Machanavajjhala: Entity Resolution: Theory, Practice & Open Challenges. PVLDB 5(12): 2018-2019 (2012)

[Huang2017] Zhipeng Huang, Nikos Mamoulis: Heterogeneous Information Network Embedding for Meta Path based Proximity. CoRR abs/1701.05291 (2017)

[Jiang2014] Yu Jiang, Guoliang Li, Jianhua Feng, Wen-Syan Li: String Similarity Joins: An Experimental Evaluation. PVLDB 7(8): 625-636 (2014)

[Karpathiotakis2016] Manos Karpathiotakis, Ioannis Alagiannis, Anastasia Ailamaki: Fast Queries Over Heterogeneous Data Through Engine Customization. PVLDB 9(12): 972-983 (2016)

[Keim2010] Daniel A. Keim, Jörn Kohlhammer, Geoffrey P. Ellis, Florian Mansmann: Mastering the Information Age - Solving Problems with Visual Analytics. Eurographics Association 2010, ISBN 978-3-905673-77-7, pp. 1-168

[Li2014] Yitong Li, Chuan Shi, Philip S. Yu, Qing Chen: HRank: A Path Based Ranking Method in Heterogeneous Information Network. WAIM 2014: 553-565

[Mann2016] Willi Mann, Nikolaus Augsten, Panagiotis Bouros: An Empirical Evaluation of Set Similarity Join Techniques. PVLDB 9(9): 636-647 (2016)

[Martinez2017] Víctor Martínez, Fernando Berzal, Juan Carlos Cubero Talavera: A Survey of Link Prediction in Complex Networks. ACM Comput. Surv. 49(4): 69:1-69:33 (2017)

[Meng2015] Changping Meng, Reynold Cheng, Silviu Maniu, Pierre Senellart, Wangda Zhang:
Discovering Meta-Paths in Large Heterogeneous Information Networks. WWW 2015: 754-764

[Olma2019] Matthaios Olma, Odysseas Papapetrou, Raja Appuswamy, Anastasia Ailamaki: Taster: Self-Tuning, Elastic and Online Approximate Query Processing. ICDE 2019: 482-493

[Papadakis2016] George Papadakis, Jonathan Svirsky, Avigdor Gal, Themis Palpanas: Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. PVLDB 9(9): 684-695 (2016)

[Shi2017] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, Philip S. Yu: A Survey of Heterogeneous Information Network Analysis. IEEE Trans. Knowl. Data Eng. 29(1): 17-37 (2017)

[Shi2019] Chuan Shi, Binbin Hu, Wayne Xin Zhao, Philip S. Yu: Heterogeneous Information Network Embedding for Recommendation. IEEE Trans. Knowl. Data Eng. 31(2): 357-370 (2019)

[Soulier2013] Laure Soulier, Lamjed Ben Jabeur, Lynda Tamine, Wahiba Bahsoun: On ranking relevant entities in heterogeneous networks using a language-based model. JASIST 64(3): 500-515 (2013)

[Wandelt2014] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, Ulf Leser: State-of-the-art in string similarity search and join. SIGMOD Record 43(1): 64-76 (2014)

[Yang2012] Yang Yang, Nitesh V. Chawla, Yizhou Sun, Jiawei Han: Predicting Links in Multi-relational and Heterogeneous Networks. ICDM 2012: 755-764

[Yu2016] Minghe Yu, Guoliang Li, Dong Deng, Jianhua Feng: String similarity search and join: a survey. Frontiers Comput. Sci. 10(3): 399-417 (2016)

[Zhan2017] Qianyi Zhan, Jiawei Zhang, Philip S. Yu, Junyuan Xie: Community detection for emerging social networks. World Wide Web 20(6): 1409-1441 (2017)