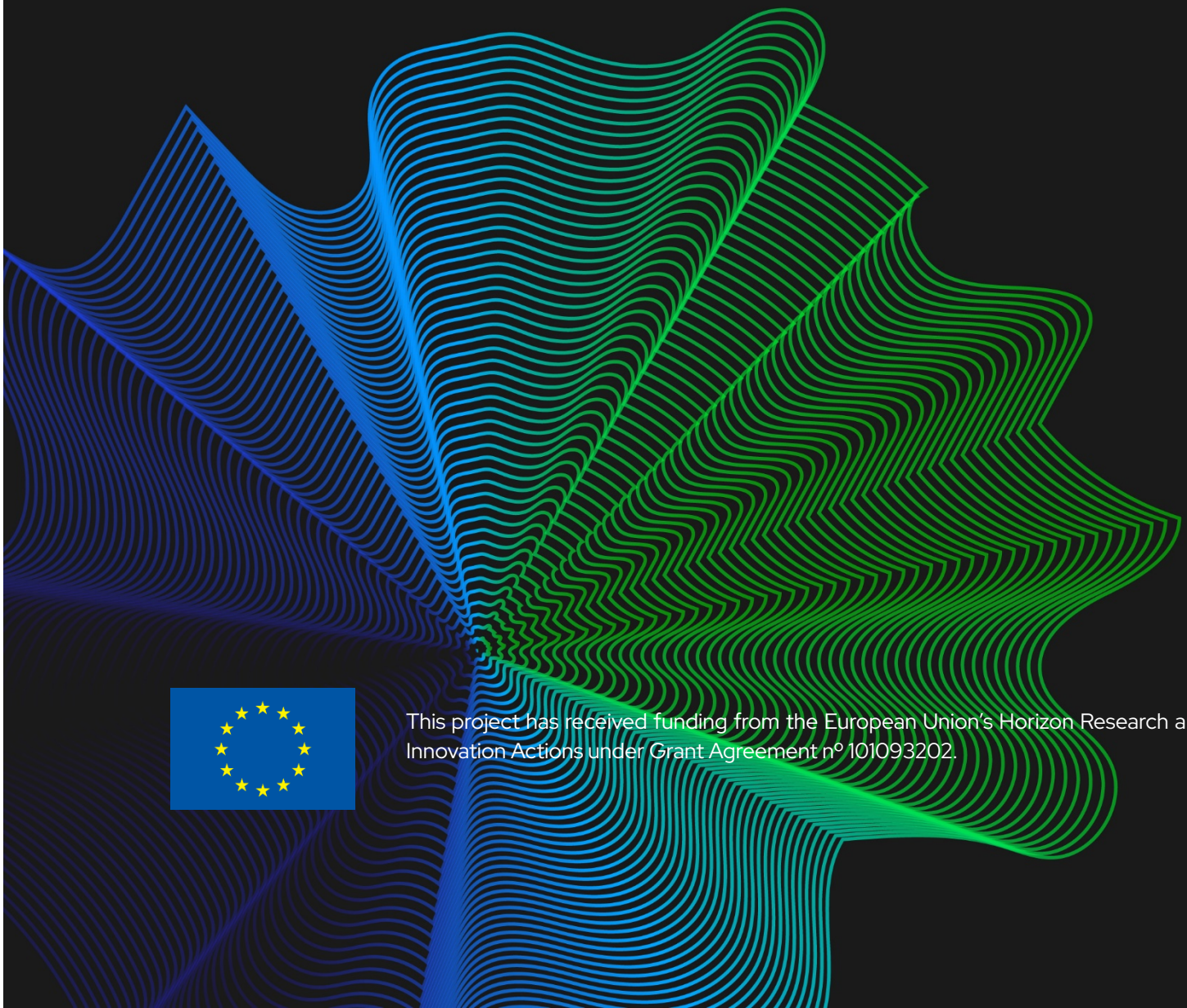
GRAPH MASSIVIZER

D3.1: MASSIVE GRAPH INCEPTION ARCHITECTURE AND DATA MANAGEMENT



This project has received funding from the European Union's Horizon Research and Innovation Actions under Grant Agreement nº 101093202.

Title	Document Version
Massive Graph Inception Architecture and Data Management	1.0

Project Number	Project Acronym	Project Title
101093202	Graph-Massivizer	Extreme and Sustainable Graph Processing for Urgent Societal Challenges in Europe

Contract Delivery Date	Actual Delivery Date	Deliverable Type* Security**
M12 (December 2023)	M12 (December 2023)	R-PU

*Type: *R – Document Report; *DMP – Data Management Plan; *DEM – Demonstrator, pilot, prototype; *Ethics: *Other.

***Dissemination Level: **PU - Public – fully open; **SEN - Sensitive – limited under the conditions of the Grant Agreement;

***EU classified –RESTREINT-UE / EU-RESTRICTED, CONFIDENTIEL-UE / EU-CONFIDENTIAL, SECRET-UE / EU-SECRET under Decision 2015/444.

Responsible	Organization	Contributing WP
Brian Elvesæter	SINTEF	WP3

Author	Organisation
Radu Prodan	AAU
Giorgio Pedrazzi	CINECA
Jože Rožanec	JSI
Wolfgang Schell, Aaron Eberhart, Peter Haase	MPH
Laurenţiu Vasiliu, Thomas Haselwanter	PER
Brian Elvesæter, Daniel Schroeder, Dumitru Roman	SINTEF
Ana Lucia Vărbănescu	UTW
Michael Cochez, Ruud van Bakel, Alexandru Iosup	VU

Abstract

This deliverable provides a research and analysis overview of basic graph operations, graph creation and synthetic data generation approaches, graph benchmark frameworks and tools, and graph sampling techniques.

Based on the research and analysis, it defines key concepts related to BGOs for the Graph-Massivizer project. A BGO is an atomic computation on a given graph. It has a unique signature and can have multiple implementations, offered as a function through a software library or a software service API.

Next, the deliverable defines the graph inception architecture, focusing on the structural composition and interactions of the Graph-Inceptor and Graph-Scrutinizer software tools. These tools will support a set of BGOs, which has been identified based on the use case and platform requirements analysis.

Finally, the deliverable describes the initial preliminary releases of the Graph-Inceptor and Graph-Scrutinizer tools.

Keywords

Architecture design; Basic graph operations; Big data; Computing continuum; Digital twin; Graph-processing; Green finance; High-performance computing; Inference; Knowledge graphs; Reasoning; State-of-the-art; Simulation; Software release; Sustainability; Synthetic data; Workflow.

REVISION HISTORY

Revision	Date	Description	Author (Organisation)
v0.1	08.09.2023	Document structure (table of contents).	Brian Elvesæter (SINTEF)
v0.2	06.10.2023	Initial requirements analysis; State-of-the-art analysis.	All authors
v0.3	20.10.2023	Updated requirements analysis; Initial architecture description.	All authors
v0.4	03.11.2023	Updated state-of-the-art analysis; Initial BGO architecture description.	All authors
v0.5	24.11.2023	Finalized state-of-the-art analysis; Requirements analysis; Updated architecture and tool sections.	All authors
v0.6	04.12.2023	Finalized architecture and tool sections; First consolidated draft for internal review.	All authors
v0.7	08.12.2023	Updated consolidated draft for internal review.	All authors
v0.8	12.12.2023	Feedback from internal reviewers.	Alexandru Iosup (VU) Giorgio Pedrazzi (CINECA)
v0.9	15.12.2023	Addressed internal review comments.	Brian Elvesæter (SINTEF)
v1.0	30.12.2023	Final review and formatting.	Haleh Dizaji (AAU) Zoha Azimi Ourimi (AAU) Radu Prodan (AAU)

This project has received funding from the European Union´s Horizon Research and Innovation under Grant Agreement nº 101093202.

More information available at https://graph-massivizer.eu/

COPYRIGHT STATEMENT

The work and information provided in this document reflect the opinion of the authors and the Graph-Massivizer project consortium and not necessarily the views of the European Commission. The European Commission is not responsible for any use of the contained information. This document and its content are the property of the Graph-Massivizer Consortium. The applicable laws determine all rights related to this document. Access to this document does not grant any right or license to the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the Graph-Massivizer Consortium and are not to be disclosed externally without prior written consent from the Graph-Massivizer Partners. Each Graph-Massivizer Partner may use this document in conformity with the Graph-Massivizer Consortium Grant Agreement provisions.

INDEX

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The Graph-Massivizer project researches and develops a high-performance, scalable, and sustainable toolset and platform for information processing and reasoning based on the massive graph representation of extreme data in general graphs, **knowledge graphs (KG)**, and **property graphs**, which integrate patterns and store interlinked descriptions of objects, events, situations, and concepts with associated semantics.

1.1 OBJECTIVE AND SCOPE

This deliverable presents the results of the research and development in work package 3 (WP3) for the three tasks: **T3.1** on massive graph creation, **T3.2** on graph queries and machine learning (ML) at an extreme scale, and **T3.4** on an interface for graph operators. Each of these tasks started in month 4 (M4) of the project and focused during the first year of the project on four areas:

- **Research and analysis of basic graph operations (BGO)**, graph creation and synthetic generation approaches, graph benchmark frameworks and tools, and distributed graph sampling techniques.
- Define a set of BGOs that perform specific computations on a graph. A BGO has a unique signature and can have multiple implementations.
- Define a **graph inception architecture**, focusing on the structural composition and the interactions of the software components that construct the Graph-Inceptor and Graph-Scrutinizer tools.
- Develop **preliminary versions of Graph-Inceptor and Graph-Scrutinizer** tools supporting an initial set of BGOs, offered as library functions or as services through an application programming interface (API).

1.2 RELATIONSHIP TO OTHER DELIVERABLES

The definition of the **graph inception architecture** extends the work presented in the deliverables D1.3 and D2.1, aligned with the deliverable D6.1.

Deliverable D1.3 [1] titled **"Graph-Massivizer Data Management Plan"** identified the planned data to collect, process, and generate by the Graph-Massivizer project, considering the data formats of the datasets in the requirements analysis of the Graph-Inceptor tool.

Deliverable D2.1 [2] titled **"Graph-Massivizer Requirements Elicitation and First Architecture Design"** defined an initial high-level architecture for the Graph-Massivizer toolkit. This deliverable refined the part of the architecture involving the Graph-Inceptor and Graph-Scrutinizer tools.

Deliverable D6.1 [3] titled **"Graph-Massivizer Infrastructure, Tools, and Use Cases"** focused on the Graph-Massivizer infrastructure comprising all tools in the Graph-Massivizer toolkit, which extended the architecture defined in the deliverable D3.1 focused on the Graph-Inceptor and Graph-Scrutinizer tools.

1.3 DELIVERABLE STRUCTURE

This deliverable contains seven sections.

Section 2 gives an overview of the research and analysis on the following topics: BGOs, data ingestion, graphs data formats, graph benchmarks, synthetic data generation, and distributed graph sampling.

Section 3 describes the requirements analysis process and the resulting technical requirements for the graph inception architecture with an initial set of supporting BGOs. The starting focus was analyzing the **use cases (UC)** and platform requirements presented in the deliverable D2.1 and the data sources described in the deliverable D1.3, included in **Annex A** for reader convenience.

Section 4 describes the graph inception architecture, starting from the architecture designed in the deliverable D2.1. An excerpt in **Annex B** describes a further architecture refinement in the Graph-Inceptor and Graph-Scrutinizer tools developed in WP3. We use an illustrative example workflow consisting of a set of BGOs to describe their interoperation.

Section 5 describes the design and implementation of the initial preliminary release of Graph-Inceptor.

Section 6 describes the architecture we envision for Graph-Scrutinizer and provides insights into the current implementation.

Section 7 concludes the report and outlines tasks for future work in WP3.

2 RESEARCH OVERVIEW AND ANALYSIS

This section overviews the state-of-the-art research and analysis on BGOs, data ingestion, graph data formats, graph benchmarks, synthetic data generation, and distributed graph sampling.

2.1 BASIC GRAPH OPERATIONS

This section briefly overviews the understanding of BGOs in the selected scientific articles. Then, it defines the key concepts related to BGOs relevant to the Graph-Massivizer project. Finally, it provides an illustrative example used in this deliverable to explain the role of BGOs in the Graph-Massivizer toolkit.

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. Several papers have introduced the concept of basic graph operations to perform operations on graph data structures effectively.

Wang et al. [4] proposed eight BGOs for graph computing: loading graph data (`Load`), counting the number of vertices (`VerticeNum`) and edges (`EdgesNum`), counting the out-degree (`VertexOutDegree`) and in-degree (`VertexInDegree`) of a specific vertex, obtaining the source (`EdgeSource`), and destination vertex (`EdgeDestination`) of a specific edge, and storing graph data (`Store`).

Leskovec et al. [5] considered `Generate`, `Save` and `Load` functions as BGOs, while Ni et al. [6] defined a BGO for convolutional graph neural networks (GNNs).

One can also consider graph algorithms, such as the Single Source Shortest Path (SSSP) algorithm, as a BGO. A multitude of graph algorithms are available in several libraries. These libraries include the Networkx[1] Python library, RAPIDS cuGraph,[2] a GPU accelerated graph analysis library, Stanford Network Analysis Project (SNAP)[3] C++/Python library for massive graph processing, Dive into Graphs (DIG)[4] turnkey library for graph deep learning research, the GAP Benchmark Suite,[5] the NVIDIA Graph Analytics library (nvGRAPH)[6] and Gunrock graph algorithms for CUDA/C++ GPU Graph Analytics.[7]

2.1.1 Definitions

Based on the literature review, we define key concepts related to BGOs.

Definition 2.1. *A **BGO** is an atomic computation on a given graph. It has a unique signature and can have multiple implementations, for example, different algorithms, programming models, or target devices. A BGO is already available at compile-time and is immutable.*

[1] https://networkx.org/
[2] https://github.com/rapidsai/cugraph
[3] https://snap.stanford.edu/snappy/index.html
[4] https://github.com/divelab/DIG
[5] https://github.com/sbeamer/gapbs
[6] https://developer.nvidia.com/nvgraph
[7] https://github.com/gunrock/gunrock/tree/main/examples/algorithms

Ideally, a BGO implements a single algorithm, but larger BGOs can also enable better optimization.

The goal of defining BGOs is twofold: (1) to allow predictable composition of workflows and (2) to enable performance modelling and prediction for both BGOs and workflows.

Definition 2.2. *A **graph processing workflow** is a combination of multiple complex BGOs that process the input graph(s) in multiple steps. The connections between BGOs are, in fact, data transfers.*

Definition 2.3. *A **BGO performance model** is a function $T(BGO(G, Imp, M))$ that, given an input graph G, a BGO implementation Imp, and a target machine M, can predict its performance (latency or throughput).*

Definition 2.4. *A **BGO energy model** is a function $E(BGO(G, Imp, M))$ that, given an input graph G, a BGO implementation Imp, and a target machine M, can predict the energy consumption of the target BGO.*

Section 3.2 presents an initial set of BGOs planned in the project.

2.1.2 UC-0: Research Collaboration Network

To better explain the role of BGOs, we present a preliminary fundamental example referred to as **UC-0**.

Description

Assuming a scholarly research collaboration network, we want to answer the following query: **"How do I reach the most popular researcher in a common field?"**. We assume the network includes information about publications (e.g., title, authors, venues) and authors' research fields. We further assume the rewarding of a "popular researcher", decided based on a metric that considers her extensive publications *and* co-authors.

We use OpenAlex,[8] a free and open catalogue of the world's scholarly research system, as the example source data for the network. It is one of multiple similar datasets provided by metaphacts that describes scholarly entities and their interconnection. The types of entities include works, authors, sources, institutions, concepts, publishers, and funders. At the time of this writing, OpenAlex contained 243 million scholarly works.

Implementation

The implementation in Figure 1 solves the query through seven BGOs.

- `LoadNetwork`: Given a dataset name and/or path, this operation ingests the data into the system as easily as connecting to a database or loading a file,

[8] https://openalex.org/about

Input: F: network; K: research topic; S: starting node;
Output: SP: shortest path from S to the most popular researcher in field K.

1: $N \leftarrow \texttt{LoadNetwork}(F)$
2: $N_K \leftarrow \texttt{Filter}(N, K)$
3: $G_K^0 \leftarrow \texttt{Data2Graph}\left(N_K, \texttt{"co}-\texttt{authorship"}\right)$
4: $G_K^1 \leftarrow \texttt{BC}\left(G_K^0\right)$ ▷ Betweenness centrality
5: $D \leftarrow \texttt{FindNode}\left(G_K^1, \texttt{"max"}\right)$
6: $G_K^2 \leftarrow \texttt{SSSP}\left(S, G_K^1\right)$
7: $SP \leftarrow \texttt{FindPath}\left(S, D, G_K^2\right)$
8: **return** SP

Figure 1: UC-0 implementation example.

depending on the input parameters. The selection of BGO implementation should require no user intervention (unless otherwise instructed);

- **`Filter`:** This operation selects the subset of data relevant to the query;
- **`Data2Graph`:** This operation transforms the **Resource Description Framework (RDF)** data format to a graph using a SPARQL query. Different implementations are possible, depending on the data storage and query choices. For this example, we connect vertices that have a co-authored paper and score their connection based on the number of co-authored publications;
- **`Betweenness Centrality (BC)`:** This operation executes the BC algorithm aiming to rank the nodes in popularity. The output is a graph with BC labels, where higher centrality indicates higher popularity.
- **`FindNode`:** This operation searches the node with a given property, for example, the node with the maximum BC score as the destination of the query.
- **`SSSP`:** This operation encodes the shortest path-finding algorithm from the source node to all other nodes.
- **`FindPath`:** This operation selects a single, minimal path, by the definition of SSSP, from a source to the destination and returns it.

Execution

Figure 2 depicts the flow across the Graph-Massivizer tools to execute this graph processing workflow. An **extract, transform, and load (ETL)** pipeline provided by **Graph-Inceptor** maps the OpenAlex source data[9] into a JavaScript Object Notation (JSON) format to the SemOpenAlex[10] [7] KG in RDF format. A series of BGOs (in red) supported by Graph-Inceptor and **Graph-Scrutinizer** process the resulting RDF graph data.

The graph processing workflow is available at this stage, and the execution engine combining Graph-Optimizer, Graph-Greenifier, and Graph-Choreographer tools is ready to deploy the computation. Firstly, **Graph-Optimizer** assesses the performance and energy consumption of different implementations of each

[9] https://docs.openalex.org/download-all-data/snapshot-data-format
[10] https://semopenalex.org/resource/semopenalex:About

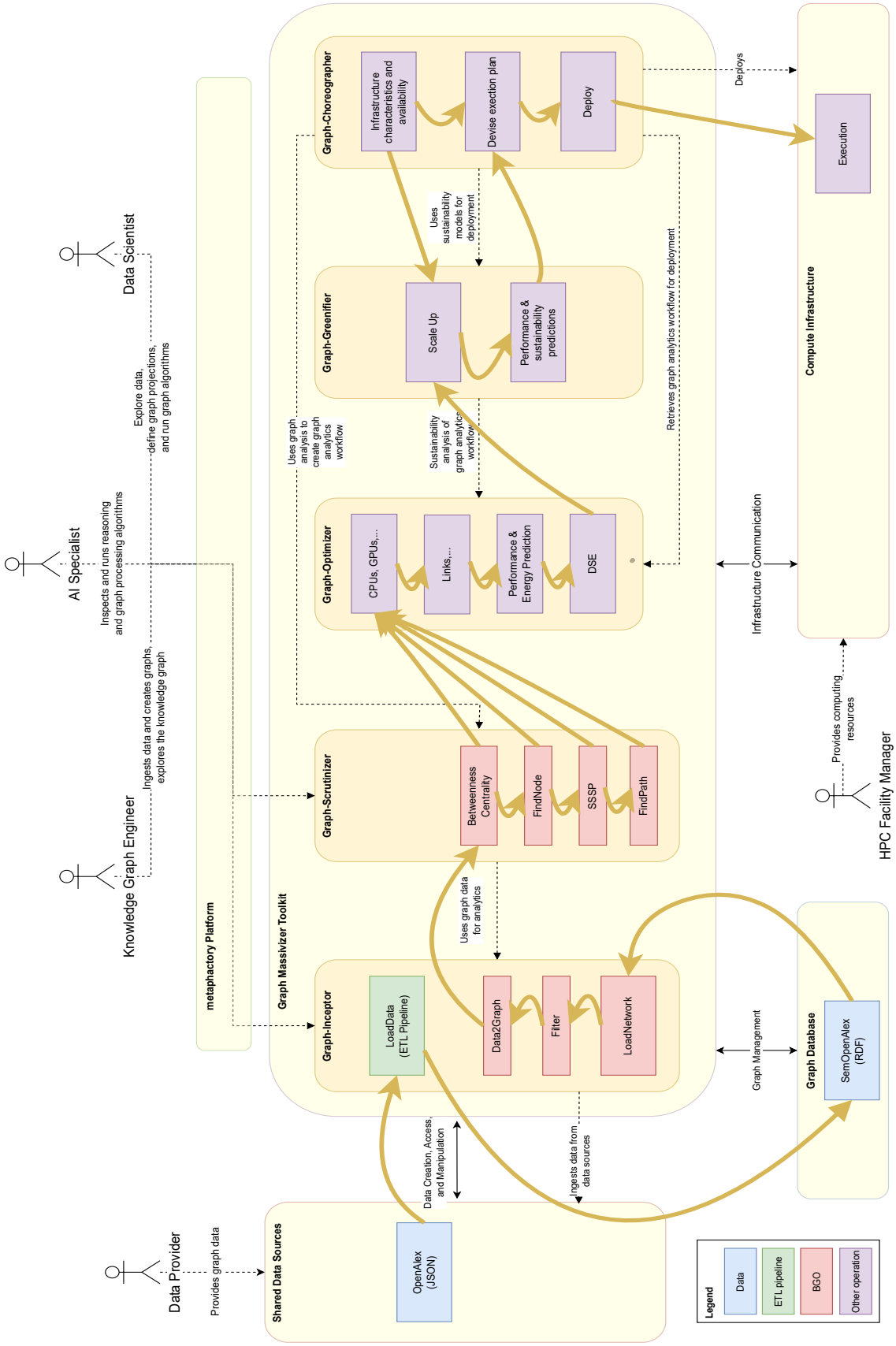


Figure 2: UC-0 flow across the Graph-Massivizer tools.

BGO and uses analytical models to estimate the performance of the entire workflow for the given graph. A design space exploration (DSE) process uses this estimation to rank and select the most promising BGO implementations and recommend hardware platforms for the workflow.

Graph-Greenifier receives these data, scales up the analysis and adds further scheduling and resource constraints to provide scale-accurate performance and sustainability predictions.

Finally, using these predicted performance data together with real information from the infrastructure, **Graph-Choreographer** devises an execution plan, deploys the workflow on the real system, collects performance metrics, and potentially adjusts (with or without further help from the Graph-Optimizer and Graph-Greenifier) the execution at runtime.

Observations

We make four observations.

1. The proposed implementation is not unique. For example, one can imagine a more complex operation where `LoadNetwork`, `Filter`, and `Data2Graph` execute as a single BGO. Similarly, the `BC` and `FindNode` collapse in a single BGO, much like `SSSP` and `FindPath`. These choices, however, are likely to create more specialized BGOs, which are not reusable for other cases, yet they could perform better in this example.
2. We intentionally did not mention any implementation or optimization aspects of these BGOs. For example, we could implement `SSSP` so that it stops when the destination has been reached, or we can implement it to be parallel, accelerated, or even approximate. However, Graph-Massivizer assumes that the framework guides and makes these choices.
3. Different implementations are available in Graph-Massivizer for each BGO, with varying hardware platforms, algorithms, and in-memory data structures and performance characteristics for different input data. Each implementation features an analytical model that, combined with the input graph data, provides predictions for performance and energy consumption. While the analytical BGO models depend on the algorithms, a (semi)automated calibration fits different platforms (e.g., CPUs, GPUs) and input graphs.
4. For predicting the performance of the entire graph processing workflow, we use the mapping of application kernels to actual hardware, potential scheduling concerns, and potential data sharing or transfers. The proposed analytical prediction model is portable to different systems.

2.2 DATA INGESTION

This section focuses on enhancing data ingestion methods for graph data processing. The goal is to research and develop an efficient data ingestion pipeline procedure, emphasizing the refinement of ETL processes. In addition, we aim to innovate smart data ingestion pipeline components, involving understanding and applying load patterns and data content patterns to optimize the ingestion.

We start with an overview of current graph streaming frameworks, assessing their strengths and weaknesses. Afterwards, we examine why these frameworks often have limited BGO support. We conclude that creating a new graph streaming framework to address these challenges, catering to the specific needs of large-scale graph data processing, is necessary.

2.2.1 Existing Graph Streaming Frameworks

Graph streaming frameworks have emerged as essential tools for handling dynamic, interconnected data in the fast-evolving data processing domain. This section analyses the current state of these frameworks, outlining their capabilities and areas for improvement.

From the general data streaming technologies, frameworks like Apache Kafka, Apache Flink, and Apache Spark provide extensible and adaptable capabilities for graph processing tasks but do not solely focus on graph data management. Given this backdrop, examining the adaptability of these platforms to graph data processing becomes crucial. While their primary design is not around graph data, their extended functionalities strengthen graph processing tasks. At the same time, their generic nature introduces limitations when repurposing for graph-specific applications. We now explore the most relevant of these frameworks in detail, emphasising how a general-purpose design influences their effectiveness in graph streaming contexts and identifying specific areas where they excel or fall short.

Apache Flink Gelly,[11] an extension of Apache Flink for graph processing, offers an API integrated with Flink's streaming engine and enabling scalable real-time graph processing. However, it lacks specific optimisations for graph processing, which limits its effectiveness in more specialized graph operations.

Apache Spark GraphX,[12] integrated within the Spark ecosystem, provides unified data processing capabilities. Still, it faces memory constraint challenges due to its in-memory computing nature suitable to batch-oriented processing, which may not align with all (real-time) graph processing needs.

JanusGraph,[13] in conjunction with Kafka, is a robust solution for scalable graph databases suitable for complex, large-scale graph applications. This combination, however, introduces complexity in operation and maintenance, especially in distributed settings. Additionally, the dual management of a graph database and streaming platform (like Kafka) can lead to performance inefficiencies.

Neo4j Streams[14] provides seamless integration with the Neo4j graph database for real-time processing. This limitation to a specific database technology affects its flexibility and poses challenges in scaling large datasets.

Amazon Neptune Streams,[15] part of the Amazon Web Services (AWS) suite, offers a managed and easy-to-operate graph database service. However, it

[11] https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/
[12] https://spark.apache.org/graphx/
[13] https://janusgraph.org/
[14] https://github.com/neo4j-contrib/neo4j-streams
[15] https://docs.aws.amazon.com/neptune/

leads to potential AWS vendor lock-in and may incur high costs for extensive or long-term use.

2.2.2 Tailored Graph Streaming Framework Innovation

Building upon the analysis of existing graph streaming frameworks, it becomes evident that platforms like Apache Flink Gelly, Apache Spark GraphX, JanusGraph with Kafka, and Neo4j Streams offer robust solutions for graph data processing but fall short in addressing specific needs vital Graph-Massivizer. This gap underscores the necessity to develop a new graph streaming framework tailored to overcome these limitations. Four arguments support this initiative.

- **Extendability:** Existing frameworks lack extendability, particularly for graph processing optimizations. While some frameworks permit the implementation of custom operators, their streams do not support extensions with optimizations tailored for graph processing. Optimizing entire pipelines in a multi-operator setting is essential for Graph-Massivizer.
- **Programming language agnostic operators:** Graph-Massivizer requires language agnostic operators supporting various operations, including GNNs. This demand stems from the need for a more inclusive and versatile framework that can cater to a broad range of programming environments and requirements, which the current frameworks fail to address adequately.
- **Category theory and functional programming:** We aim to establish correct streams according to category theory, deeply anchored in functional programming principles. This approach will ensure a more rigorous and mathematically sound foundation for handling graph data streams with greater precision and reliability. The existing frameworks do not entirely conform to these theoretical principles to the extent Graph-Massivizer requires.
- **Multiple graph storage formats and database connectors:** Existing frameworks do not support multiple graph storage formats and database connectors. Graph-Massivizer needs a seamless interface with various in-memory graph formats and a connection to different graph database technologies.

In summary, while the current graph streaming frameworks provide foundational capabilities for graph data processing, they do not entirely meet the specialized requirements of Graph-Massivizer for large-scale graph data processing. Thus, the research and development (R&D) of a novel graph streaming framework extendable for graph-specific optimizations, supporting programming language agnostic operators, adhering to principles of category theory in functional programming, and accommodating multiple graph storage formats and database connectors is imperative.

2.3 GRAPHS FORMATS

The choice of graph formats directly impacts the effectiveness of data ingestion, particularly their alignment with the ETL processes. By understanding this relationship, we can better tailor the data ingestion methods to suit the specific characteristics of the chosen graph formats, enhancing the overall efficiency.

Furthermore, integrating graph formats into the broader data ingestion strategy helps assess their compatibility with the proposed graph streaming framework (see Section 5.3), including their influence on the performance and flexibility of the system, especially in terms of extendability and support for various graph storage formats and database connectors.

According to the Graph-Massivizer architecture defined in the deliverable D2.1, RDF is a key supporting format, providing the foundation for the (virtual) KG. Section 4.2 provides a further description of the KG creation, including different RDF serialization formats in Section 4.2.8.

In this section, we provide an overview of additional graph formats possibly required by `Data2Graph` BGOs that transform one (graph) data format to another, expected as input for another BGO (e.g., an algorithm such as SSSP).

2.3.1 Edge List

An edge list is a simple and common data structure representing a graph as a collection of nodes (or vertices) and edges. In an edge list, each row or record represents a single edge in the graph or the connection between two nodes.

In its simplest form, an edge list contains two columns: one for each edge endpoint. For example, Figure 3 shows a simple undirected graph with three nodes and three edges. Each row represents an edge. The first row indicates an edge between nodes 1 and 2, the second row between nodes 2 and 3, and the third row between nodes 3 and 1. An undirected graph typically represents each edge twice, once for each order of the nodes (not illustrated in Figure 3).

```
1 2
2 3
3 1
```

Figure 3: Simple edge list example.

Variations of the edge list can include more information.

Weighted edge list

In many graphs, edges have weights, typically representing the strength or capacity of a connection. A weighted edge list includes a new column for the edge weight. For example, Figure 4 indicates that the edge between nodes 1 and 2 has a weight of 0.5, the edge between nodes 2 and 3 has a weight of 1.2, and the edge between nodes 3 and 1 has a weight of 0.8.

```
1 2 0.5
2 3 1.2
3 1 0.8
```

Figure 4: Weighted edge list example.

Directed edge list

In directed graphs (DGs) or digraphs, edges have a direction from one node to another (but not in the opposite direction). In an edge list for a DG, the order of the nodes matters: the first node is the source, and the second node is the target. Figure 5 represents a DG with three edges: from node 1 to node 2, from node 2 to node 3, and from node 3 to node 1.

```
1 2
2 3
3 1
```

Figure 5: Directed edge list example.

Edge list with attributes

In some graphs, edges have additional attributes represented by more columns in an edge list. For example, in a social network, edges might have a 'relationship' attribute, such as the 'friend', 'coworker', and 'family' attributes in Figure 6.

```
1 2 friend
2 3 coworker
3 1 family
```

Figure 6: Social network example.

Advantages

- **Flexibility**: Combining these variations is possible. For example, one can have a weighted, directed edge list with additional attributes.
- **Simplicity**: Edge lists are arguably the simplest way to represent a graph. They are easy to understand and implement.
- **Space efficiency for sparse graphs**: For sparse graphs with fewer edges than nodes, edge lists can be more space-efficient than adjacency matrices.
- **Easy add and removal of edges**: Edge lists make it easy to add or remove edges, involving adding or removing rows from the list.

Disadvantages

- **Slow neighbour access**: To find all neighbours of a particular node, one has to scan through the entire edge list, which can be slow for large graphs.
- **Slow edge existence check**: Checking if a specific edge exists can be slow for the same reason.
- **Undirected graphs redundancy**: An undirected graph typically includes duplicate representation of edges for the different order of nodes.

2.3.2 Adjacency List

An adjacency list associates each node with a list (or collection) of its neighbouring nodes directly connected via an edge. Figure 7 depicts an example where each row starts with a node, followed by a colon and a list of adjacent nodes. For example, the first row indicates that node 1 connects nodes 2 and 3.

```
1: 2, 3
2: 1, 3
3: 1, 2
```

Figure 7: Adjacency list example for undirected graphs.

Variations of the adjacency list can include more information.

Weighted adjacency list

In a weighted graph, edges have weights included in the adjacency list, for example, as values 0.5, 0.8, and 1.2 in Figure 8.

```
1: (2, 0.5), (3, 0.8)
2: (1, 0.5), (3, 1.2)
3: (1, 0.8), (2, 1.2)
```

Figure 8: Adjacency list example for weighted graphs.

Adjacency list for directed graphs

In a DG, each edge has a direction. An edge from node 1 to node 2 includes 2 in the adjacency list of 1, but not vice versa. as depicted in Figure 9.

```
1: 2
2: 3
3: 1
```

Figure 9: Adjancey list example for DGs.

Adjacency list with edge attributes

In some graphs, adjacency lists include additional edge attributes. For example, Figure 10 illustrates attributes such as 'friend', 'family', and 'coworker' joined to node identifiers.

```
1: (2, friend), (3, family)
2: (1, friend), (3, coworker)
3: (1, family), (2, coworker)
```

Figure 10: Adjacency list example with edge attributes.

Advantages

- **Flexibility**: Combining these representations, such as a weighted adjacency list for a DG with edge attributes, is possible. The representation depends on the specifics of the graph and required operations.
- **Space efficiency**: Adjacency lists are more space-efficient than adjacency matrices for representing sparse graphs with fewer edges than nodes, as they only store the existing edges.
- **Fast neighbour access**: Adjacency lists allow for quick access to all neighbours of a node, which can be beneficial in algorithms that iterate them.
- **Edge attributes**: Adjacency lists can easily store additional attributes for each edge, such as weights.

Disadvantages

- **Slow edge existence check**: In an adjacency list, checking if a specific edge exists between two nodes can be slow, as it may require a linear scan of a node's neighbour list.
- **Data structure complexity**: Compared to edge lists and adjacency matrices, adjacency lists are a more complex data structure, harder to implement and manipulate.
- **Undirected graph redundancy**: An undirected graph represents each edge twice in an adjacency list, once for each endpoint node.

2.3.3 Graph Modeling Language

Graph modeling language (GML)[16] represents graphs in a hierarchical ASCII-based file format. Its key features include simplicity, flexibility, and the ability to incorporate arbitrary properties and attributes within graph, node, and edge definitions. GML represents graphs, nodes, and edges as key-value pairs. Figure 11 shows a graph with two nodes and one edge. Each node has an `id` and a `label`. Two `source` and `target` nodes define an edge.

GML is extensible in several ways to include additional information.

1. **DGs**: Edges have a direction in a DG. GML does not have a built-in concept of DGs but can represent them by adding a `direction` attribute to the edges.
2. **Weighted graphs**: In a weighted graph, edges have weights, included in GML by adding a `weight` attribute.
3. **Additional attributes**: GML allows adding arbitrary attributes to graphs, nodes, and edges. For example, a `colour` attribute to nodes represents their colour in visualisation.

[16] https://gephi.org/users/supported-graph-formats/gml-format/

```
graph [
  node [
    id 1
    label "Node 1"
  ]
  node [
    id 2
    label "Node 2"
  ]
  edge [
    source 1
    target 2
  ]
]
```

Figure 11: Simple undirected graph GML example.

GML does not strictly enforce the type or range of attribute values, so the software must read the GML file and interpret them correctly.

Advantages

- **Simplicity**: GML is straightforward and easy to understand. The language structure is easy to read and write, even for humans, not just for machines.
- **Flexibility**: GML allows arbitrary properties and attributes and is extendable to various graph structure representation and their associated metadata.
- **Popularity**: Many graph software tools support GML, enabling easy importing and exporting of graphs represented in GML across different platforms.

Disadvantages

- **Verbosity**: GML files can be larger than other graph file formats because of their verbose nature, especially for massive graphs, leading to increased storage and memory requirements and slower processing times.
- **Loose schema**: GML does not strictly enforce the type or range of attribute values. The software is responsible for correctly reading and interpreting the GML file, which can lead to inconsistencies.
- **Limited built-in features**: GML cannot represent some common graph concepts, like directed or weighted edges and requires custom attributes.

2.3.4 Graph Exchange XML Format

Graph exchange XML format (GEXF)[17] is an extensible, easy-to-use, and open-standard file format for exchanging graph structures and their associated data.

[17] https://gexf.net/

GEXF defines both the structure of the graph (nodes and edges) and the metadata associated with these elements (e.g., attributes, dynamic properties).

Graph is the root element containing all other elements. It has an attribute called `defaultedgetype`, which defines the type of edges in the graph (e.g., directed, undirected).

Nodes and edges are the basic elements of the graph. Each node has a unique `id` and a `label`. Edges connect nodes defined by their `source` and `target` node `ids`. Edges can also have an `id`, `label`, and `weight`.

Attributes store additional data associated with the nodes and edges. Each attribute has an `id`, `title`, and `type`.

Dynamic networks that change over time are expressible using the `start` and `end` attributes on nodes and edges, as well as on attribute values.

Figure 12 shows a GEXF example of a simple DG with two nodes and one edge. Node 0 connects to node 1 via a directed edge.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
<graph mode="static" defaultedgetype="directed">
    <nodes>
        <node id="0" label="Node 0" />
        <node id="1" label="Node 1" />
    </nodes>
    <edges>
        <edge id="0" source="0" target="1" />
    </edges>
</graph>
</gexf>
```

Figure 12: GEXF example.

Advantages

- **Expressiveness**: GEXF is highly expressive, allowing for the representation of complex graph structures with attributes, hierarchies, and dynamics.
- **Popularity**: Many graph software tools support GEXF, allowing easy exchange of graph data between different systems.
- **XML format**: GEXF is relatively easy to generate, parse and process using standard XML tools and libraries.

Disadvantages

- **Verbosity**: Like other XML-based formats, GEXF can be verbose, especially for large graphs, leading to increased storage and memory requirements and slower processing times.

- **Complexity**: While GEXF's expressiveness is advantageous for complex features, it makes the format more complex to understand and use than simpler formats like edge or adjacency lists.
- **Limited adoption**: Despite being a general-purpose graph exchange format, GEXF is not as widely adopted as other formats like GraphML.

2.3.5 GraphML

GraphML[18] is a comprehensive, friendly file format for graphs. It encodes graph structures in XML and provides a flexible mechanism to add application-specific data. GraphML can model directed, undirected, and mixed graphs, hypergraphs, hierarchical graphs, graphs with ports, and attributes with user-defined types.

Figure 13 shows a simple, undirected graph in GraphML with two nodes represented by a `node` tag with an `id` attribute and one edge represented by an `edge` tag with `source` and `target` attributes referring to their node `ids`.

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <edge source="n0" target="n1"/>
  </graph>
</graphml>
```

Figure 13: GraphML example.

The GraphML format supports various extensions:

1. **Attributes** for elements like graphs, nodes, and edges with user-defined data types.
2. **Hypergraphs** with edges connecting more than two nodes.
3. **Hierarchical graphs**: through nested graphs support.
4. **Graphs with ports**: for connecting edges to specific ports on nodes rather than to the nodes themselves.

Advantages

- **Expressiveness**: GraphML is highly expressive in representing complex graph structures, including directed, undirected and mixed graphs, hypergraphs, hierarchical graphs, and graphs with ports.
- **Attribute support**: GraphML supports attributes for elements like graphs, nodes, and edges with user-defined data types.
- **Popularity**: Many graph software tools support GraphML, allowing easy exchange of graph data across different systems.

[18] https://graphml.graphdrawing.org/

- **XML format**: GraphML is relatively easy to generate and parse using standard XML tools and libraries.

Disadvantages

- **Verbosity**: Like other XML-based formats, GraphML can be verbose, especially for large graphs, leading to increased storage and memory requirements and slower processing times.
- **Complexity**: While GraphML's expressiveness is advantageous for complex features, it makes the format more complex to understand and use compared to simpler formats like edge lists or adjacency lists.
- **Limited adoption**: Despite being a general-purpose graph exchange format, GraphML did not receive as wide adoption as other formats like GEXF.

2.3.6 GraphBLAS

GraphBLAS[19] project aims to create a specification for building graph algorithms in the language of linear algebra. It expresses various graph algorithms as a small set of highly effective operations for processing large-scale graph data on sparse matrices. These operations are not straightforward, but they represent building blocks to construct complex graph algorithms once implemented efficiently. The GraphBLAS specification includes the definition of:

- Sparse matrix data types;
- Set of operations on these data types, like multiplication, addition, etc.
- Operations to manipulate these data types, like extract, assign, etc.

The implementation of the GraphBLAS specification includes various languages such as C, C++, and Python.

GraphBLAS variations

The GraphBLAS specification is not a software library but a blueprint for creating efficient libraries for graph computation. Several libraries following the GraphBLAS specification exist:

- **SuiteSparse:GraphBLAS**[20] is a complete implementation of the GraphBLAS specification in the C language;
- **GraphBLAS Template Library (GBTL)** is a C++ library for GraphBLAS;
- **pyGraphBLAS** is a Python interface to GraphBLAS, implemented on top of the SuiteSparse:GraphBLAS library;
- **GraphBLAST** is a GraphBLAS implementation optimised for GPUs.

[19] https://graphblas.org/
[20] https://people.engr.tamu.edu/davis/GraphBLAS.html

Advantages

- **Expressiveness**: GraphBLAS provides a powerful expressive framework for building graph algorithms. Expressing these algorithms in the language of linear algebra simplifies their implementation and understanding.
- **Efficiency**: GraphBLAS works with sparse matrices, which are a common representation for large-scale graph data, leading to substantial performance gains, especially for large graphs.
- **Portability**: GraphBLAS specification is language-agnostic, implemented in various languages such as C, C++, and Python. Thus, GraphBLAS algorithms are portable across different programming languages and platforms.
- **Reusability**: GraphBLAS provides reusable building blocks for constructing graph algorithms, saving time and effort in implementing correct algorithms.

Disadvantages

- **Learning curve**: To use GraphBLAS effectively, developers must understand graph algorithms and linear algebra, requiring a significant learning curve.
- **Efficient implementations**: While the GraphBLAS specification defines the interface for graph operations, their efficiency depends on the specific implementation. Developing an efficient implementation of GraphBLAS is a complex task that requires expertise in sparse matrix operations.
- **Limited dynamic graph support**: GraphBLAS is highly efficient for static graphs but may not be as efficient for dynamic graphs with frequently changing edges and vertices.

2.4 GRAPH BENCHMARKS

This research designs and implements benchmarks for smart ETL, BGOs, distributed graph sampling and streaming graph ML algorithms. The development of benchmarks is of primary importance to drive progress in a specific research field, providing a common research goal and methodological aspects that guarantee common pitfalls are avoided and that the research is comparable. In particular, developing graph benchmarks ensures using graphs of the same characteristics, considering the same set of high-performance baselines and metrics used to evaluate the results across a wide range of research [8].

Designing the benchmarks around real-life scenarios and datasets is a good practice [9]. A comprehensive graph benchmark must consider the key batch and streaming paradigms of graph analysis [10] and multiple aspects of graph processing, such as data loading and required transformation. By analyzing existing graph benchmarks, we found no benchmark to measure how graph samplers behave in a distributed and streaming environment.

The next sections overview existing graph benchmarks and outline the research in Graph-Massivizer on a comprehensive benchmark for smart ETLs, BGOs and graph sampling for streaming and distributed environments.

2.4.1 Existing Graph Benchmarks

We performed a comprehensive literature review on graph benchmarks, analyzing more than 400 papers. We briefly describe in this section 22 graph benchmarks developed and used over the years in the graph community. For each benchmark, we describe the key components, datasets, tasks, and metrics considered and provide references to a relevant paper, website and code repository, where available.

Table 1 lists the benchmarks chronologically. In particular, we notice a lack of benchmarks related to graph sampling methods (only two [11, 12]). Furthermore, while not specifically targeting ETL procedures, many benchmarks consider data ingestion and graph-building tasks, along with performance metrics.

High Performance Computing Scalable Graph Benchmark

Purpose	Compact application with multiple analysis techniques (kernels) accessing a single weighted DG
Key components	• Data generator (of edge tuples with start vertex, end vertex, directed edge weight) • Kernels (for task execution)
Data	Synthetic data
Tasks	Four kernels: • data generation • large sets classification • graph extraction • graph analysis (computing BC)
Metrics	Traversed edges per second
References	[13]

Table 1: Graph benchmarks.

Year	Benchmark	Acronym	Reference
2009	HPC[a] Scalable Graph Benchmark		[13]
2010	Graph500		[14]
2012	XGDBench		[15]
2013	Waterloo Graph Benchmark	WGB	[16]
2013	LinkBench		[17]
2013	Graph Database Becnhmark	GDB	[18]
2014	BigDataBench		[19]
2014	BigOP		[20]
2014	Benchmark for Social Media Analytics	BSMA	[21]
2015	GAP Berkeley	GAP	[22]
2015	GraphBIG		[23]
2015	Sparkbench		[24]
2016	Linked Data Benchmark Council Graphalytics	LDBC	[25]
2018	Graph Based Benchmark Suite	GBBS	[26]
2018	GraphTides		[27]
2019	GARDENIA		[28]
2019	UniBench		[29]
2020	Open Graph Benchmark	OGB	[8]
2021	Non-Homophily-Large-Scale Benchmark		[30]
2022	GraphWorld		[31]
2022	Large-Scale GCN[b] Benchmark		[32]
2023	Temporal Graph Benchmark	TGB	[33]
2023	Relational Deep Learning Benchmark	RelBench	[34]

[a] HPC: High Performance Computing, [b] GCN: Graph Convolutional Neural Network

Graph500

Purpose	Push industry to build machines that benefit the new large-scale graph data analysis problems, which needs to be economically important to motivate industry R&D investments
Key components	Data generator based on a Kronecker graph
Data	Six synthetic datasets (toy, mini, small, medium, large, and huge) • emulating cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks
Tasks	Graph construction, sampling 64 search keys, BFS[a], and SSSPs
Metrics	Traversed edges per second
References	Graph500 webpage[b], [14], [35]

[a] BFS: Breadth-First Search, [b] https://graph500.org/?page_id=12

XGDBench

Purpose	Graph database benchmarking platform operating in cloud service and exascale cloud infrastructures
Key components	• Data workload generator • Third-party databases (representing the target of the benchmarks) • Cytoscape (external tool for community analysis)
Data	Synthetic data
Tasks	Five categories of read/write tasks: • update heavy, read mostly, read-only, read latest, short ranges
Metrics	Average throughput workload
References	[15]

Waterloo Graph Benchmark

Purpose	Objectively compare graph processing platforms using six representative graph algorithms and many real-world and synthetic datasets
Key components	• Data generator (for query execution) • Workload specification (for performance evaluation)
Data	Synthetic data
Tasks	Three types of tasks: • online queries (read-only, interactive, with low complexity, not iterative) • updates (interactive, adding or removing vertices or edges that change the graph's structure, parameters or topology) • iterative queries (complex long analytics, possibly requiring multiple passes over the data, including non-real-time updates)
Metrics	Execution time
References	[16]

LinkBench

Purpose	Realistic and challenging test for persistent storage of social and web service data, filling a gap in the available tools for researchers, developers, and administrators
Key components	• Linkbench driver • Graph generator • Workload generator • Graph store adapter • Graph store
Data	Synthetic, plausible data
Tasks	Four operations: • point reads for associations and objects identified by primary key • create, delete, and update operations for associations and objects identified by primary key • association range queries for a given identifier, type, and timestamp range, ordered from latest to oldest • association count queries, for the number of visible out-edges of a given type from a given node
Metrics	• CPU usage (user, system, idle, wait) • Read and write I/O (operations per second, rate in MB/s) • Resident memory size • Persistent storage size (including temporary indices, tables, and logs)
References	GitHub repository[a], [17]

[a] https://github.com/facebookarchive/linkbench

Graph Database Becnhmark

Purpose	Compare distributed graph databases, focusing on four important players in the graph databases market (Neo4j, OrientDB, Titan and DEX)
Key components	• GDB statistics module • GDB operational module (which runs the benchmark) • Graph database
Data	Synthetic data considering a scale-free network generator based on the Barabasi-Albert model
Tasks	Three types of tasks: • load workload (start a graph database and load it with a particular dataset) • traversal workload (perform a particular traversal on the graph (e.g., shortest path workload, neighbourhood exploration workload)) • intensive workload (synchronised parallel clients send together a specific request (e.g., get vertices or edges by identifier or property, update a property on vertices, add edges between two vertices))
Metrics	Execution time (as perceived on the client side)
References	[18]

BigDataBench

Purpose	Diversity of data and workloads for evaluating big data systems and architectures
Key components	• Data generator (structured, semi-structured, and unstructured data) • Data processing stack (built on top of Hadoop) • Micro-architecture level profiling tools (Perf) • Analytics software to understand the workloads (MPICH2 and Rubis)
Data	Synthetic data and six real-world data sets: • Wikipedia entries • Amazon movie reviews • Google web graph • Facebook social network • e-commerce transaction data • ProfSearch person resumes
Tasks	19 big data workloads grouped into three categories of applications (online service, real-time analytics, and offline analytics). The graph-related tasks are PageRank and connected components.
Metrics	• User-perceivable (requests per second, operations per second, data processed per second) • Architectural (MIPS, cache MPKI)
References	[19]

BigOP

Purpose	End-to-end system benchmarking framework featuring the abstraction of representative operation sets, workload patterns, and prescribed tests to assist in choosing the right system for big data processing
Key components	• Test generator (specification of a subset of operations and processing patterns, dataset, workload generation method, and metrics) • System under test (generated tests from prescriptions)
Data	Synthetic graph-related data (random DGs)
Tasks	PageRank
Metrics	Execution time (measured on client side)
References	[20]

GAP Berkeley

Purpose	Standardised graph processing evaluations to: • alleviate methodological issues • prevent common evaluation mistakes
Key components	Input graphs and GAP benchmark suite: • large real-world and synthetically generated graphs with billions of edges • graph-embedded domain-specific language • workload characterization • graph kernels • best practices evaluation (timing methodologies and optimizations)
Data	16 real-world datasets[a]: • SuiteSparse matrix collection, Texas A&M[b] • SNAP, Stanford[c] • laboratory for web algorithmics • road networks, DIMACS grand challenge #9[d] • graphs to partition, DIMACS grand challenge #10 • Twitter, KAIST • Twitter, INRIA • AltaVista web graph, Yahoo! • social networks, Max Planck Institute • Facebook, CURRENT lab @ UCSB • Wikipedia (2010) link graph • Open web crawl • Microsoft Academic graph • Koblenz network collection • Google+ • web data commons – hyperlink graphs
Tasks	Six graph-processing tasks: • BFS (direction optimizing) • SSSPs (delta stepping) • PageRank (iterative method in pull direction) • connected components (Afforest & Shiloach-Vishkin) • BC (Brandes) • triangle counting (order invariant with possible relabelling)
Metrics	Average execution time (of tasks executed multiple times)
References	GAP webpage[e], GitHub repository[f], [22]

[a] http://gap.cs.berkeley.edu/datasets.html, [b] https://sparse.tamu.edu/, [c] http://snap.stanford.edu/, [d] https://www.diag.uniroma1.it/challenge9/, [e] http://gap.cs.berkeley.edu/benchmark.html, [f] https://github.com/sbeamer/gapbs

Benchmark for Social Media Analytics

Purpose	Benchmark data management systems supporting analytical queries over social media, considering synthetic and real-world data
Key components	• Data feeding component • Workload generator • Performance testing toolkit
Data	Synthetic data, along with a real-life dataset crawled from Sina Weibo, the most popular microblogging service in China
Tasks	Graph operations, temporal, aggregate, and hotspot queries
Metrics	Latency, throughput and scalability over data volume
References	[21]

GraphBIG

Purpose	Understand the impact of modern graph computing on the hardware architecture and enable future architecture and system research
Key components	Five architectural levels: • hardware, database, middleware, analytics, visualization
Data	• Synthetic (LDBC graph) • Real-world data (Twitter graph, IBM knowledge repo, IBM Watson gene graph, and CA road network)
Tasks	Four types of tasks: • graph traversal (BFS, DFS) • graph construction, update, and topology morphing • graph analytics (shortest path, k-Core decomposition, connected components, triangle count, graph coloring, Gibbs inference) • social analysis (degree centrality, BC)
Metrics	• CPU-based (cache MPKI[a], branch and cache miss rates, first-level DTLB[b] penalty, frontend stall, backend stall, retiring, bad speculation) • GPU-based (branch and memory divergence rates)
References	[23]

[a] MPKI: Misses per kilo instructions, [b] DTLB: Data translation lookaside buffer

Sparkbench

Purpose	Spark-specific benchmarking suite guiding the design and optimization of Spark data analytic platforms.
Key components	• Data generator • Spark analytic platform
Data	Synthetic data, controlling the size of the input datasets to finish each workload within 2 to 15 minutes
Tasks	Four types of workloads • ML (logistic regression, support vector machine, and matrix factorization) • graph computation (PageRank, SVD++, and triangle count) • SQL queries (Hive, RDD[a] relation) • streaming applications (Twitter, PageView)
Metrics	• CPU time (user, system, and IO waiting) • Memory (used, cached, swap) • Disk IO (read, write, read local, write local) • Streaming applications (Twitter, PageView) • Network IO (receive, send) • Average job execution time • Workload execution time
References	[24]

[a] RDD: Resilient distributed dataset

GraphTides

Purpose	Systematic system development, performance measurements, engineering, and comparisons of stream-based graph systems
Key components	Open, platform-agnostic, extensible framework for representative, versatile workloads, comprising: • graph stream generator • graph stream replayed • runtime metrics logger • streaming rate
Data	Synthetic data
Tasks	Six standard graph algorithms on workloads characterized by: • stream composition (event mix and interleaving) • topology changes (direction, types, locality and temporal distribution) • state changes (types, locality and temporal distribution, update behaviour) • social analysis (degree centrality, BC)
Metrics	• Level 0 metrics (CPU load, memory usage, network I/O, and disk I/O) • Level 1 metrics (internal throughput rates and communication latencies between workers) • Streaming metrics (actual ingress rate) • Computation metrics (latency, accuracy, correctness)
References	[27]

Linked Data Benchmark Council Graphalytics

Purpose	Objectively compare graph processing platforms using: • six representative graph algorithms • many real-world and synthetic datasets
Key components	• Benchmark executor (verifies the setup, formats and loads the graph, executes and archives the benchmarks, and deletes the graph) • Benchmark runner (runs the benchmark considering four phases: startup, run, validate, and finalize)
Data	51 datasets, split into eight sizes (2XS, XS, S, M, L, XL, 2XL, 3XL)
Tasks	Six graph processing tasks, executed three times (single node or distributed): • BFS • PageRank • weakly connected components • community detection using label propagation • local clustering coefficient • SSSPs
Metrics	Load time, makespan, mean processing time, edges per second, edges and vertices per second, three-year TCO[a], and price-per-performance
Special process	LDBC renews every few years. The last renewal occurred in 2020 [36]
References	LDBC webpage[b], GitHub repository[c], [37]

[a] TCO: Total cost of ownership, [b] https://ldbcouncil.org/benchmarks/graphalytics/, [c] https://github.com/ldbc/ldbc_graphalytics

GARDENIA

Purpose	Emerging irregular workloads (widely employed in graph analytics, sparse linear algebra, and ML) for future accelerator architecture research
Key components	No specific components highlighted in the paper
Data	Eight graphs (cage14, Freescale, Flickr, Web-Google, Youtube, LiveJ, Soc-Pokec, Wikipedia) and seven sets of graphs: • SuiteSparse matrix collection, Texas A&M • 9th & 10th DIMACS implementation challenge[a] • SNAP • the Koblenz network collection[b] • network data repository[c] • Real-World input graphs[d]
Tasks	Nine graph processing tasks: BFS, SSSPs, BC, PageRank, connected components, triangle counting, SGD[e], sparse matrix-vector multiplication, symmetric Gauss-Seidel smoother, vertex coloring
Metrics	Instructions per cycle, single instruction, multiple threads utilization, branch divergence rate, memory divergence rate
References	GitHub repository[h], [28]

[a] https://sites.cc.gatech.edu/dimacs10/, [b] http://konect.uni-koblenz.de/, [c] https://networkrepository.com/index.php, [d] http://gap.cs.berkeley.edu/datasets.html, [e] SGD: Stochastic Gradient Descent, [h] https://github.com/chenxuhao/gardenia

Graph Based Benchmark Suite

Purpose	Suite of scalable, provably efficient implementations of over 20 fundamental graph problems for shared-memory multicore machines
Key components	Library with no particular components highlighted
Data	Three datasets: com-Orkut, Hyperlink2012, and Hyperlink2012-Sym

Tasks	20 fundamental graph processing tasks:
	• clustering problems:
	  ◦ SCAN graph clustering, graph-based hierarchical agglomerative clustering
	• connectivity problems:
	  ◦ low-diameter, decomposition, connectivity, spanning forest, biconnectivity, MST[a], strongly connected components, covering problems
	• graph coloring:
	  ◦ maximal matching, maximal independent set, approximate set cover
	• eigenvector problems:
	  ◦ PageRank
	• substructure problems:
	  ◦ triangle counting, approximate densest subgraph, k-Core (coreness), degeneracy ordering (low-outdegree orientation), k-clique counting, 5-cycle counting, k-truss
	• shortest path problems:
	  ◦ unweighted SSSP (BFS), general weight SSSP (Bellman-Ford), integer weight SSSP (weighted BFS), single-source BC, single-source widest path, k-spanner

Metrics	Execution time
References	GBBS webpage[b], GitHub repository[c], [38]

[a] MST: Minimum Spanning Tree, [b] https://paralg.github.io/gbbs/, [c] https://github.com/ParAlg/gbbs

Unibench

Purpose	Benchmark for a holistic and rigorous evaluation of the performance and usability of multi-model databases
Key components	• Scalable multi-model data generator • Set of workloads including multi-model aggregation, join, and transaction
Data	Synthetic datasets, considering five data models
Tasks	12 workloads consisting of a set of complex read-only queries and read/write transactions involving at least two data models and covering different • business cases (i.e., individual, conversation, community, commerce) • technical perspectives (i.e., choosing the right join type and order performing complex aggregation, ensuring consistency and efficiency)
Metrics	Query time, throughput
References	[29]

Open Graph Benchmark

Purpose	Automated end-to-end graph ML pipeline to simplify and standardize: • graph data loading • experimental setup • model evaluation

Key components	• OGB data loader
	  ◦ download, process, store, and return requested dataset objects
	• OGB evaluator
	  ◦ for standardized and reliable evaluation

Data	19 datasets, categorized into three sizes (small, medium, large) and four challenges:
	• node property prediction[a]
	  ◦ ogbn-products, ogbn-proteins, ogbn-arxiv, ogbn-papers100M, ogbn-mag
	• link property prediction[b]
	  ◦ ogbl-ppa, ogbl-collab, ogbl-ddi, ogbl-citation2, ogbl-wikikg2, ogbl-biokg, ogbl-vessel
	• graph property prediction
	  ◦ ogbg-molhiv, ogbg-molpcba, ogbg-ppa, ogbg-code2
	• very large-scale graph ML datasets
	  ◦ MAG240M, WikiKG90Mv2, WikiKG90Mv2, PCQM4Mv2

Tasks	Three prediction tasks: node, link, graph properties
Metrics	Accuracy, AUC[c], Hits@100, Hits@50, MRR[d], AP[e], F1 score, and MAE[f]
References	OGB webpage[g], GitHub repository[h], [8], [39]

[a] https://ogb.stanford.edu/docs/nodeprop/, [b] https://ogb.stanford.edu/docs/linkprop/, [c] AUC: Area Under the ROC Curve, [d] MRR: Mean Reciprocal Rank, [e] AP: Average Precision, [f] MAE: Mean Absolute Error, [g] https://ogb.stanford.edu/, [h] https://github.com/snap-stanford/ogb

Non-Homophily-Large-Scale Benchmark

Purpose	Collection of nonhomophilous datasets from various application areas to foster GNNs for node classification beyond the homophily regime
Key components	• Datasets • Two node classification methods
Data	Pen94, Pokec, genius, twitch-gamers, snap-patents, arxiv-year, wiki
Tasks	Node classification
Metrics	Accuracy and AUC
References	GitHub repository[a], [30]

[a] https://github.com/CUAI/Non-Homophily-Large-Scale

GraphWorld

Purpose	Mitigate common issues across GNN benchmarks, such as inadequate generalization, incremental overfitting, and unscalable development
Key components	• Data generators (for GNN test dataset simulations using attributed-graph and label generators) • Model execution (requiring a list of models and hyperparameter exploration settings, task formulation, metrics, and datasets to sample from the data generators).
Data	Synthetic datasets
Tasks	Three types of tasks: • node classification • link prediction • graph property prediction using the following algorithms: ARMA, APPNP, FiLM, GAT, GATv2, GCN, GIN, GraphSAGE, SGC, SuperGAT, Transformer, LinearRegression, MLP[a], PageRank, LinkPrediction
Metrics	• AUC (for node classification and link prediction) • Scaled mean-squared error (for graph property prediction)
References	[31]

[a] MLP: Multilayer Perceptron

Large-Scale Graph Convolutional Neural Network Benchmark

Purpose	Systematically formulate the representative methods of large-scale graph training into several branches and further establish a fair and consistent benchmark for them by a greedy hyper-parameter search
Key components	• Greedy hyperparameter search (for sampling, precomputing, and label propagation) • GCN implementations
Data	Three datasets: Flickr, Reddit, and ogbn-products
Tasks	Two types of tasks: • sampling and precomputing (GraphSAGE, FastGCN, LADIES, ClusterGCN, GraphSAINT, SGC, SIGN, SAGN, GAMLP, C&S, AdaGCN, EnGCN) • label propagation
Metrics	GPU memory usage, throughput, and convergence
References	GitHub repository[a], [32]

[a] https://github.com/VITA-Group/Large_Scale_GCN_Benchmarking

Temporal Graph Benchmark

Purpose	Automated ML pipeline for reproducible temporal graph research, including data loading, experimental setup and performance evaluation
Key components	No documented components
Data	Two task-based dataset types: • dynamic link property prediction (tgbl-wiki-v2, tgbl-review-v2, tgbl-coin, tgbl-comment, tgbl-flight) • dynamic property prediction (tgbn-trade, tgbn-genre, tgbn-reddit, tgbn-token)
Tasks	Two prediction tasks: • dynamic link property prediction • dynamic property prediction
Metrics	Mean reciprocal rank, normalized discounted cumulative gain
References	TGB webpage[a], GitHub repository[b], [33]

[a] https://tgb.complexdatalab.com/, [b] https://github.com/shenyangHuang/TGB

Relational Deep Learning Benchmark

Purpose	Benchmark datasets driving research on relational deep learning
Key components	• Data (loading, predictive task specification, and (temporal) data splitting) • Model (graph transformation, GNN model building) • Evaluation (standardized protocol given a file of predictions)
Data	Two datasets: • Amazon product review e-commerce database • Stack exchange question-and-answer website database
Tasks	Prediction tasks: • user lifetime value • user churns • active user site contribution • number of upvotes received by a question
Metrics	MAE, average precision
References	Relbench webpage[a], GitHub repository[b], [34]

[a] https://relbench.stanford.edu/, [b] https://github.com/snap-stanford/relbench

2.4.2 Graph-Massivizer Benchmarking Architecture

Based on the insights obtained from the described benchmarks, we propose an architecture for creating benchmarks addressing smart ETLs, BGOs, distributed graph sampling and streaming graph ML algorithms. The architecture depicted in Figure 14 has seven components.

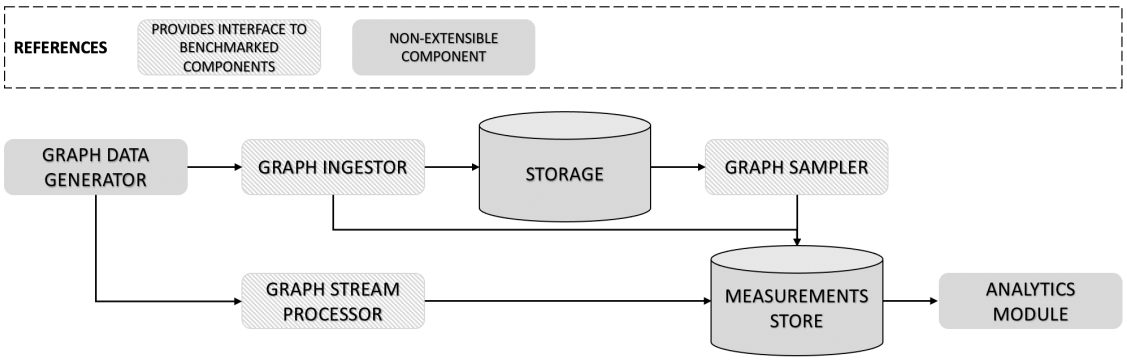


Figure 14: Graph-Massivizer benchmarking architecture.

- **Graph data generator** creates synthetic data pushed to the ingestor to store it in graph storage.
- **Graph ingestor** receives input data and forwards it in graph format to a given graph storage.
- **Storage** saves graphs in a persistent format.
- **Graph sampler** performs sampling on graphs retrieved from the storage.
- **Graph stream processor** executes streaming graph ML models.
- **Measurements store** collects measurement data from three components: ingestor, graph sampler and stream processor.
- **Analytics module** analyses the measurements stored in the measurement store to obtain desired metrics.

Three components expose plugin interfaces to particular benchmarking methods developed by the research community: (1) ingestor (to benchmark smart ETLs), (2) graph sampler (to benchmark sampling algorithms), and (3) stream processor (to benchmark streaming graph ML algorithms).

2.5 SYNTHETIC DATA GENERATION

This section presents the initial research conducted in WP3 on methods for synthetic graph data generation, covering traditional mathematical and heuristic methods and deep generative models.

2.5.1 Synthetic Data

It is common in many scientific and engineering simulations in domains like bioinformatics, climate analysis, or social sciences to use synthetic data to test and

analyse functional and non-functional properties. Such generators are, therefore, domain-specific and not reusable. Graph-Massivizer needs data generators constructed for both tool development (for testing and benchmarking purposes) and for the specific requirements of the UC-1 targeting synthetic financial data. Thus, we focus on data generation in the form of graphs.

2.5.2 Synthetic Graphs

Several mathematical-based graph generators are common for the analysis of graph algorithms.

Graph generation

Graph500[21] uses the Kronecker model to generate a graph using the Kronecker product recursively. Each multiplication exponentially increases the size of the graph. The Kronecker graph generative model makes a given network denser over time, while the diameter shrinks [40].

R-MAT [41] offers a fast generation of realistic directed, undirected, and bipartite graphs [42] following a power-law degree distribution by operating on the adjacency matrix. PaRMAT [43] can generate large R-MAT graphs and provide controllability over certain properties (e.g., directed, undirected, cyclic).

LDBC Datagen [44] is a synthetic graph data generation tool for the LDBC Social Network Benchmark [45]. Datagen graphs mimic real-world social media graphs following a discretised power-law degree distribution. Datagen produces evolving, time-correlated graphs, modelling various types of connections between entities (e.g., friendship, message posting). It also includes several different degree distributions.

Graph scaling

While R-MAT, Kronecker, or Datagen generate graphs from scratch, an alternative is to scale existing graphs, preserving some of their potentially unique properties. Using multiple scale factors, one can obtain a family of correlated graphs. Such scaling methods can produce more diverse graphs by allowing users to provide arbitrary input graphs.

GScaler [46] creates a larger graph by decomposing an initial graph into small pieces, subsequently scaled, based on the input and output degree correlation of nodes and edges. GScaler aims to preserve the properties of the original graph but does not provide any control over them. Furthermore, it lacks support in preserving undirected graphs.

Musketeer [47] is a newer graph generation method which considers an input graph along with parameters such as node growth and edit rates and aims to reproduce its features [42] by applying coarsening and un-coarsening operations to obtain the desired scale. While this algorithm shows promising results, the existing implementation is not fast enough for large networks.

[21] http://www.graph500.org

ReCoN [48] aims to improve Musketeer's performance while preserving its quality by creating disjoint graph copies up to the preferred size. A randomization step on the edges inside and between communities connects the copies, leading to a scaled output graph.

In [49], the authors propose a heuristic for building graph families with controlled properties, starting from a given, real-life network. The method packaged as a graph scaling tool provides several parameterized ways to control graph properties and generate graph families and models to understand the graph properties of the scaled datasets.

Graph-Massivizer research

We plan to use these generators, especially graph scaling, for the performance analysis and engineering of BGOs. However, most of these generators, except DataGen, focus on graph generation and its topology and ignore vertex and edge properties or heterogeneous data, which is fundamental for KGs.

For generating such graphs for the purpose of benchmarking and testing (including the UC-1 workflow described in Section 2.5.3), we may consider open-source[22] tools like PyGraft [50] or GraphGen [51].

Finally, deep neural networks significantly advanced in graph generation, as presented in a recent survey [52]. Among these models and methods, an especially successful solution uses GraphRNN [53] for a proof-of-concept prototype implementation for data generation in UC-1.

2.5.3 Domain-Specific Financial Data Generation

Despite the generality of the tools presented in Section 2.5.2, they are difficult to tweak for the specific UC. Instead, some of them are useful as modules of a necessary novel workflow for financial data generation. Specifically, we identified five steps to generate relevant synthetic financial data.

1. Detect patterns and correlations in the original historical data, which will become part of the rules set required for generating synthetic data.
2. Handle missing data from historical data.
3. Transform the time series historical data from XML/CSV to RDF graph format.
4. Use the generated RDF graph for the synthetic graph generation.
5. Implement and execute quality rules at synthetic data generation.

We detail these steps in the next sections.

Patterns and correlations' detection

It is of utmost significance for synthetic data to incorporate analogous patterns and correlations with the ones observed in the historical sample data to be use-

[22] https://github.com/nicolas-hbt/pygraft

ful and meaningful. Such identified patterns and correlations from the original dataset will become active rules for the synthetic data creation.

Various alternatives are suitable for pattern detection in time series data, depending upon its characteristics and objectives. Each alternative has its own advantages and disadvantages (see deliverable D6.1, Annex). We will decide upon the most suitable ones during implementation.

Missing data handling

Missing data stands out as a prevalent and frustrating issue during the processing of financial data for testing, modelling, correlation analysis, and other activities essential for validating financial models. A range of factors may cause the absence of data, including manual entry errors, uncompleted fields, equipment malfunctions, data loss, or unavailability.

Synthetic data seeks to eradicate this issue entirely, enabling the utilization of complete datasets for many testing purposes and ML systems with high sensitivity to data quality (see deliverable D6.1, Annex).

GraphRNN data generation

Figure 15 presents the Graph-Massivizer approach for generating the synthetic financial data in five steps.

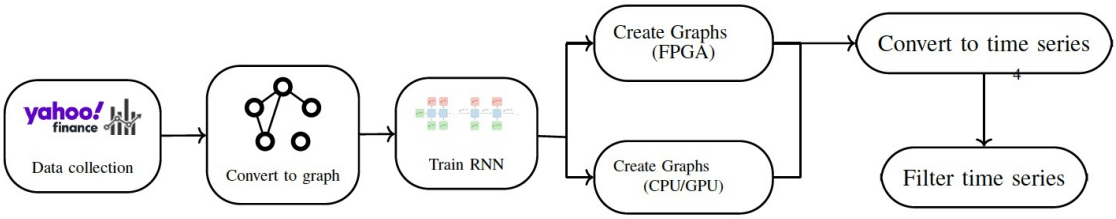


Figure 15: Synthetic financial data generation using NN.

1. **Data collection** in CSV format for training the neural network (NN).
2. **Graph conversion (BGO)** of the raw data. A survey of methods to support this transformation [54] indicates that correlation graphs obtained by calculating the correlation matrix of a set of time series are a good fit. Specifically, we correlate stock data between companies and create a matrix containing all the company correlations for a certain stock indicator. We use the method in [55] to convert the correlation matrix to an adjacency matrix of a graph by thresholding with a value τ. We experimented with different threshold values and found $\tau = 0.8$ as a good fit. We collect all these graphs (based on different time series) into a complete training dataset.
3. **NN training (BGO)**: using the GraphRNN proposed in [53] for graph generation. We further use the maximum mean discrepancy as a method to evaluate graph likeness, which requires a training procedure using the datasets created from the historical data.

4. **New graphs creation (BGO)**: using the trained RNN model generated as encoded adjacency matrices. Some additional pruning might be necessary to ensure the connectivity of the generated graphs.
5. **Time-series conversion (BGO)** from the graphs. Different procedures are also available for this step, requiring exploring the best trade-off between performance and relevance.
6. **Data filtering (BGO)** using additional quality rules to discard further generated data that is not sane. This step can potentially merge with the previous step if the conversion and filtering execute more efficiently as a fused BGO.

We implemented a proof of concept of this method[23] with promising graph generation results, while alternatives for the final conversion and rule-based pruning are still under development.

Quality rules implementation

The information supplied by PER is the primary set of instructions crucial for examining, validating, and organizing synthetic data ranges. These instructions formulated based on required global pre-conditions for synthetic data generation include rules for fundamental and technical financial data, patterns, and structures identified in the original historical data samples. They are important during synthetic data generation, ensuring that the resultant data adheres to the specified criteria.

We identified four important preconditions for producing synthetic data.

1. **Computational efficiency** requires algorithm complexity analysis, code optimisation and performance monitoring, comprising the following metrics:
   - **Maximum computation time** for generating a complete set of synthetic data under 15 hours to allow daily production.
   - **Working RAM**, **power consumption**, **external storage**, and **total cost per TB**, determined during prototype testing.
2. **Computational robustness** requires algorithm analysis, code optimisation and performance monitoring to cope with errors during synthetic data generation, unexpected or erroneous input, management of extreme data volumes of data without slowdown or crashes, maintenance of data consistency and protection against security threats and vulnerabilities.
3. **Data consistency and quality** employs AI-generated data and compare models[24] using a dedicated Python library on external resources, along with financial data-specific rules.
4. **Standardisation** requires synthetic data generation in the most common formats used by financial data, XML and CSV.

Further details on quality rules for synthetic data are available in the deliverable D6.1 (Annex).

[23] https://github.com/Mathi18R/code
[24] https://mltechniques.com/2023/09/19/new-python-library-to-evaluate-ai-generated-data-and-compare-models/

2.5.4 Synthetic Data Storage And Streaming

PER will generate synthetic data in batches, streamed and stored to the PER platform. The streamed batches will train and test financial algorithms stored with third-party repositories. For this purpose. a close synchronisation between the Graph-Massivizer and PER platforms for data storage, streaming and ingestion will be necessary.

2.6 GRAPH SAMPLING TECHNIQUES

Graphs offer a flexible approach to modelling connected systems. However, the flexibility of graphs often leads to scenarios where accessing or using the full graph is infeasible. In such scenarios, sampling [56] is a popular remedy, allowing for estimating properties of full graphs impossible to use.

Sampling can be useful for ML tasks, effectively trained on smaller fractions of the data. Several methods [57, 58, 59, 60, 61] exist, specifically tailored to GCNs [62], while more flexible approaches are possible too [63, 64].

Table 2 : Graph Samplers.

	Node-Wise	Layer-Wise	Layer-Dependent	Architecture Independent	Adaptive	Uses Feature Data	Distributed Version
GraphSAGE	✓						✓
FastGCN		✓					
LADIES		✓	✓				✓
GraphSAINT				✓			
ClusterGCN				✓			
AS-GCN		✓	✓		✓	✓	
GCN-BS	✓				✓	✓	
PASS	✓				✓	✓	

2.6.1 Node-Wise Sampling

Node-wise samplers yield a new graph sample for every layer of a GCN, connected according to these samples.

GraphSAGE [57] takes a node-wise approach that samples new nodes at every layer in the GCN. Thus, the sample size is not constant throughout the network, suffering from an exponentially growing sample size throughout the layers. Nevertheless, it can still offer scalability for more shallow networks.

2.6.2 Layer-Wise Sampling

Like their node-wise counterpart, layer-wise samplers yield a sample for each GCN layer. These methods, however, keep a fixed sample size.

FastGCN [58] is a node-wise sampler that keeps a fixed sample size throughout the GCN. FastGCN does not condition sampling in the current layer on the sample from the previous layer, often yielding to largely disconnected layers and inhibiting efficient training.

LADIES [59] uses a so-called layer-dependent sampling by keeping a fixed sample size and high connectivity between the GCN layers. This method effectively combines the strengths of GraphSAGE and FastGCN.

2.6.3 Subgraph-Based Sampling

Subgraph-based samplers differ from the previous methods by sampling once per training iteration instead of once per layer per iteration. Decoupling the sampling and the GNN makes subgraph-based samplers suitable to many settings beyond GCN training (e.g., other types of GNNs).

ClusterGCN [65] partitions the entire graph into clusters based on some graph partition algorithms (e.g., METIS) and then selects several clusters to form a batch. As a subgraph-based method, it only samples once per training step.

GraphSAINT [63] samples a subset of the nodes based on a sampling strategy and then induces the corresponding subgraph as a batch. It generates one subgraph per training step and changes all the adjacency matrices accordingly. Keeping the connections between each layer the same simplifies the sampling mechanism and allows easier adaptation to different neural architectures.

2.6.4 Graph Pre-Processing

SGC [66] leverages the node representations aggregated with K hops and feeds the resultant features to a fully connected layer.

SIGN [67] concatenates features from different hops and then fuses them as the final node representation via a linear layer.

SAGN [68] adopts an attention mechanism to combine feature representations from K hops.

2.6.5 Distributed Sampling

While sampling is a valuable tool for processing large graphs, it still has limitations. For example, it can cause significant overhead during training [69]. GPUs can speed up the sampling process, but they have limited memory and copying the full graph defeats the purpose of sampling in most cases. There are, after all, graphs that do not even fit in typical CPU memory [70].

Distributed graph sampling is a family of methods that tackles the problem of memory limitation using several machines concurrently. This setup allows split-

ting the graph into smaller parts and distributing them over several machines that could fit on them, even if the original graph could not.

Recently, several algorithms for distributed sampling on graphs emerged in the GNN area. This development is not unexpected, as training GNNs can be a slow process, making fast algorithms desirable.

Distributed graph sampling with Spark [71], while not specifically tailored towards GNNs, proposes nine distributed algorithms covering node-based, edge-based and topological sampling methods. The results show that distributed sampling can cause considerable overhead on smaller graphs but can also be more performant on larger ones.

Communication-efficient sampling for distributed GCN training [72] argues that information sharing among machines creates a bottleneck and that common graph sampling methods (e.g., GraphSAGE, FastGCN, Ladies, GraphSAINT) lack a distributed setting. The method takes a distributed sampling setting and adjusts the sampling rates by favouring samples within the cluster on the current machine (GPU), decreasing the communication overhead. A correction term introduced during the GNN training removes the introduced bias.

Distributed matrix-based sampling for GNN training [73] introduces a matrix multiplication-based distributed algorithm providing two benefits. Firstly, it can model several of the previous graph sampling algorithms (e.g. GraphSAGE, LADIES). Secondly, several architectures for efficient distributed matrix (and tensor) operations already exist, extended to other sampling schemes.

FastSample [69] offers two novel approaches for accelerating distributed GNN training for billion-scale graphs. Firstly, it implements a twice as fast kernel for DGL [74]. Secondly, it proposes to split only the feature matrix that is typically much larger than the very sparse adjacency matrix, allowing a (sparse) copy of the full adjacency matrix on every machine, thus yielding a 1.5 times increase in speed. Splitting up the feature matrix also allows for processing larger graphs.

3 REQUIREMENTS ANALYSIS

This section describes the requirements analysis process and the resulting technical requirements for the graph inception architecture, which consists of an initial set of BGOs to support. We started by analysing the UC and platform requirements presented in the deliverable D2.1. This deliverable's data sources described in **Annex A** include the analysed UC and platform requirements for reading convenience.

3.1 REQUIREMENTS ANALYSIS PROCESS

The Graph-Massivizer requirements definition methodology follows a four-step iterative process, comprising (1) requirements elicitation, (2) requirements analysis and specification, (3) requirements validation and (4) requirements evolution, according to the roadmap principles presented in [75].

Deliverable D2.1 presented a detailed set of UC and Graph-Massivizer platform requirements resulting from the requirements elicitation phase (see **Annex A**). WP3 continued this analysis to specify technical requirements for the Graph-Inceptor and Graph-Scrutinizer tools, resulting in an initial set of BGOs by following the procedure in [75]:

- Identify key findings (e.g., needs, pain points, technical setup, best practices, research challenges, technological constraints, etc.);
- Correlate any key finding to a functionality/feature/need/risk/issue;
- For any functionality/feature/need/risk/issue, derive a BGO requirement.

3.2 REQUIREMENTS SPECIFICATION

This section presents an initial set of BGOs that need support from the Graph-Inceptor and Graph-Scrutinizer tools.

3.2.1 Identified Use Case Basic Graph Operations

Based on this requirements analysis, we derived an initial set of BGOs relevant to the UCs, requiring support from the Graph-Inceptor and Graph-Scrutinizer tools. For each BGO, Table 3 shows a mapping to the corresponding platform and UC requirements.

Table 3 : Identified UC BGOs

BGO	Description	Tool	Requirement
`LoadNetwork`	Load graph data	Graph-Inceptor	`GI-1`
`Filter`	Filter graph data	Graph-Inceptor	`GI-1`
`Data2Graph`	Convert various source formats from files into a graph format	Graph-Inceptor	`GI-2`, `UC-1.1.1`, `UC-1.1.2`, `UC-2.1`, `UC-3.1.1`, `UC-3.1.2`, `UC-4.1.1`, `UC-4.1.2`, `UC-4.1.3`
`Interface2Graph`	Convert various source file formats from various sources into a graph format	Graph-Inceptor	`GI-2`, `UC-2.1`, `UC-4.1.1`, `UC-4.1.2`, `UC-4.1.3`
`StoreGraph`	Store graph data in a graph database	Graph-Inceptor	`GI-1`, `UC-1.1.1`, `UC-1.1.2`, `UC-2.1`, `UC-3.1.1`, `UC-3.1.2`, `UC-4.1.1`, `UC-4.1.2`, `UC-4.1.3`
`Graph2File`	Convert graph data into various file formats	Graph-Inceptor	`UC-4.2`
Entity similarity	Batch	Graph-Scrutinizer	`UC-1.2`
Anomaly detection	Batch	Graph-Scrutinizer	`UC-1.2`
Standing queries (indicators)	Batch, Stream	Graph-Scrutinizer	`UC-1.2`, `UC-1.3`
Pattern recognition	Batch	Graph-Scrutinizer	`UC-1.5`
Link prediction (inference)	Batch	Graph-Scrutinizer	`UC-1.3`
Graph embedding	Batch	Graph-Scrutinizer	`UC-3.4`
Link prediction (analysis)	Batch	Graph-Scrutinizer	`UC-2`
Node prediction (analysis)	Batch	Graph-Scrutinizer	`UC-2`
Causality mining	Batch	Graph-Scrutinizer	`UC-2.2`
Approximate inference	Batch	Graph-Scrutinizer	`UC-1.3`
Synthetic graph generation	Batch	Graph-Scrutinizer	`UC-1.4`, `UC-1.6`
Graph statistics	Batch	Graph-Scrutinizer	`UC-1-4`
Graph sampling	Batch	Graph-Scrutinizer	`UC-3.5.1`
Graph queries (for GUI)	Batch	Graph-Scrutinizer	`UC-4.2`
Graph clustering	Batch	Graph-Scrutinizer	`UC-4.3.1`
Node classification	Batch	Graph-Scrutinizer	`UC-4.3.1`
Causality maps	Batch	Graph-Scrutinizer	`UC-4.3.1`

3.2.2 Additional Considered Basic Graph Operations

The list in Table 3 is **not** exhaustive and may include additional BGOs discovered during the UC development. Furthermore, Graph-Inceptor and Graph-Scrutinizer may offer many other general graph and algorithms library functions or services, as already identified in the illustrative example). Table 4 lists additional BGOs considered to support.

Table 4 : Additional considered BGOs.

Category	BGO	Tool
Algorithms (level 1)	• Vertex count • Edge count • Degree distribution • Filtering • Random sampling • Neighbour iteration	Graph-Scrutinizer
Algorithms (level 2)	• Traversal BFS • Traversal DFS • Page Rank • SSSP • Diameter calculation • Triangle counting • Sampling: TIES • MST • Maximum Flow	Graph-Scrutinizer
Algorithms (level 3)	• Connected components • Weakly/Strongly connected components (in DGs) • BC • All-pairs shortest paths • Isomorphism • Summarization • Sampling: Forest Fire • Graph colouring • Community detection	Graph-Scrutinizer
Conversions (Table 3)	• File to in-memory • In-memory to in-memory • Time-series to graphs • Graphs to time-series	Graph-Inceptor
ML	• GNN model training • GNN model inference	Graph-Scrutinizer

4 ARCHITECTURE AND DATA MANAGEMENT

This section describes the **graph inception architecture**, starting from the design presented in the deliverable D2.1. We describe a further refinement of the original architecture included in **Annex B** focusing on the Graph-Inceptor and Graph-Scrutinizer tools developed in WP3. We use an illustrative example workflow of BGOs to describe how they interoperate.

4.1 GRAPH INCEPTION ARCHITECTURE

Figure 16 shows the initial graph inception architecture as a version of the Graph-Massivizer toolkit architecture container diagram (see Figure 23 in Annex B) that focuses on the Graph-Inceptor and Graph-Scrutinizer tools omitting details of the other tools and environments.

The following sections focus on two architectural aspects depicted in Figure 17:

1. **KG creation:** Section 4.2 describes a blueprint for KG creation. It defines an ETL pipeline that reads source data from the `Data Provider`, maps it to a (virtual) KG and stores the resulting KG in a graph database.
2. **Workflow definition and deployment:** Section 4.3 discusses an illustrative graph processing workflow (i.e., UC-0 with the seven BGOs) introduced in Section 2.1.2 supported by the Graph-Inceptor and Graph-Scrutinizer tools on a computing infrastructure. We replaced the term "workload" in the original diagram with "workflow" (according to the definition in Section 2.1.1).

4.2 KNOWLEDGE GRAPH CREATION BLUEPRINT

This blueprint describes various aspects for creating a KG from source data. Section 5.2 describes a concrete blueprint implementation based on an ETL pipeline consisting of an opinionated set of tools running in a public cloud.

4.2.1 Source Data

A KG may be constructed with data from many different sources, considering a few aspects regarding various types, formats, and partitioning of source data.

Structured versus unstructured data

While a KG may stem from manually created data (e.g., using form-based authoring), in most cases, data is imported from existing data sources. The data sources can be of two main types: structured and unstructured.

Structured data includes machine-readable files and formats like JSON, XML, CSV or data obtained from a database, following a well-defined syntax and may optionally have schema information (e.g. relational databases). This data type is easily readable and accessible using ready-made libraries and parsers.

Unstructured data includes documents such as emails, PDF files, images, and scanned paper documents, which generally do not make the contained infor-

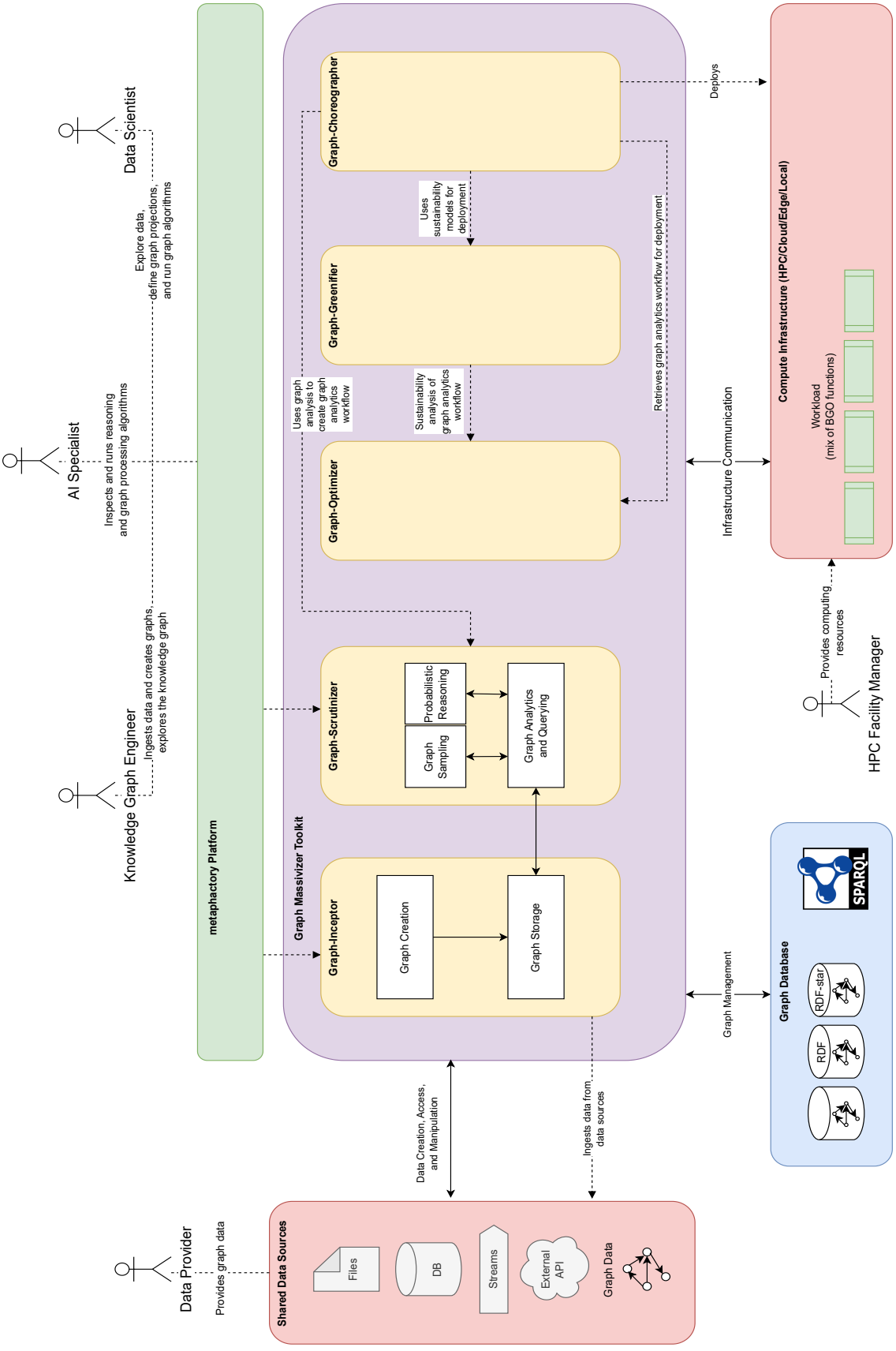


Figure 16: Graph inception architecture (level 2).

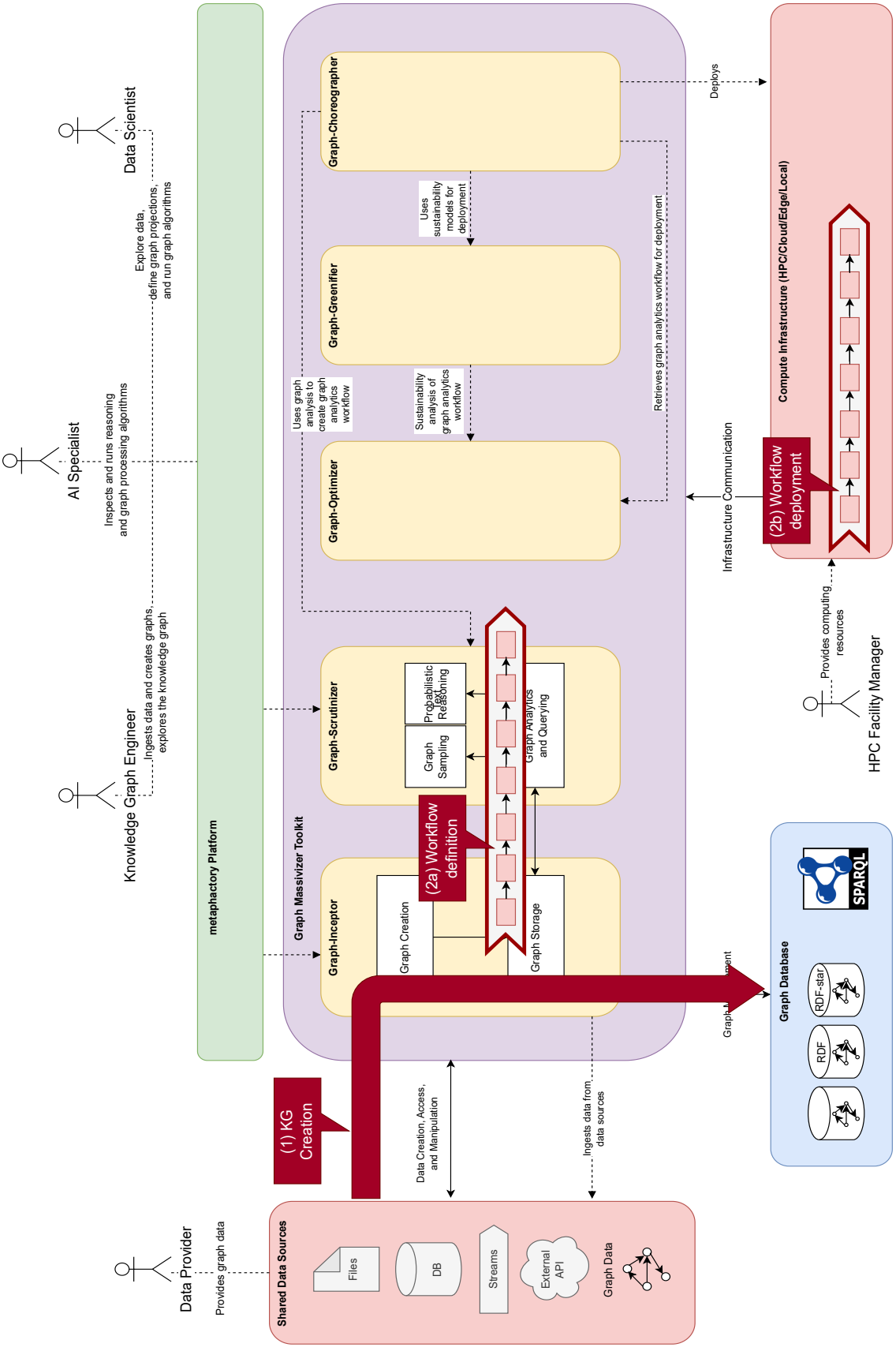


Figure 17: Graph inception architecture (annotated).

mation available in a machine-readable format. Reading, parsing, understanding, and converting unstructured data is difficult, often approached using ML techniques like optical character recognition, name entity recognition, text classification, or entity linking using services or libraries like SpaCy.[25]

This blueprint considers only structured data.

Source data formats

Structured data may reside in files and streams in many formats. Examples of widely used formats are CSV/TSV (tabular), XML (hierarchical), JSON/JSONL (hierarchical), and query results from relational databases (tabular). Other structured data formats are possible but require custom adapters or parsers.

4.2.2 Graph Data Model

Graphs are represented using RDF[26] as a sequence of statements (or triples), each describing a fact, such as an attribute of a resource, edge or node, or a relation between two resources.

RDF-star[27] is an extension of RDF and supports expressing statements on statements, allowing to model edges with attributes. RDF-star can express any graph and be a bridge to and from Labeled Property Graphs (LPG).

4.2.3 Data Partitioning

Real-world big datasets may be too large to treat as a single unit. Partitioning divides data into workable chunks based on different criteria, depending on the UC, such as data source or origin, application domain (i.e., entity types), licensing, visibility, security or size.

Source data partitioning

Converting an entire big dataset to RDF in a single operation is impossible, so splitting it into manageable pieces, such as individual files, is necessary for scalable ingestion. Integrating data from a database requires exporting it into a set of files with an appropriate amount of data per conversion unit with a maximum size. This operation moves the paging of data and partitioning outside of the conversion process, as it depends on the structure and amount of data and additional considerations like (de-)normalization, number and size of files, portion of the dataset to handle, and others.

Named graphs partitioning

Partitioning the source data is mainly relevant for the RDF conversion. In addition, data also needs partitioning within the KG. Partitioning graph data generates so-called named graphs, identified using an internationalized resource

[25] https://en.wikipedia.org/wiki/SpaCy
[26] https://www.w3.org/TR/rdf11-concepts/
[27] https://www.w3.org/2021/12/rdf-star.html

identifier (IRI), which extends the part triple subject-predicate-object with the named graph IRI and forms a quadruple. Each named graph has further descriptive metadata, including type, description, or license information.

Splitting graph data into named graphs follows different criteria, such as named graph per data domain (i.e., entity type), named graph per source, or a mix of them, depending on the UC and application data. The data size is typically not a criterion for data partitioning within a KG.

4.2.4 Knowledge Graph Creation Methods

Creating a KG needs approaches optimized for performance or loose coupling.

Materialization

The common approach to KG creation is to materialize RDF in a graph database. Materialization involves retrieving source data from the original source, converting it to RDF if required and persisting it (e.g., in a graph database). A fully materialized KG provides the best performance for querying by storing data in a native graph format and within a graph database optimized for graph querying and processing. Materialization works especially well for highly connected data, exposing relationships from data integrated from multiple sources.

Virtualization

Virtualization leaves data at its original store in its original format, such as tabular data in a relational database. Querying the KG retrieves the data from the original data source and transforms the results to RDF on the fly. Virtualization leads to loose coupling by transferring and converting data to RDF only upon request, with no need to maintain two data stores simultaneously. This approach is suitable for rarely-required data and follows a tabular structure like time series or other mass data types. As a drawback, query performance is low due to on-the-fly data extraction and conversion.

Federation

Federation is a variant of virtualization combining confidential or sensitive data with public information KG for specific purposes and allowing owners to maintain their original data. For example, federation can combine data from public datasets like Wikidata[28] with internal data or from multiple sources. The SPARQL 1.1 query language[29] provides a built-in feature for expressing information needs over multiple RDF KGs. SERVICE clauses[30] can combine information from remote RDF sources with data from the local RDF database.

4.2.5 Knowledge Graph Creation

Creating a KG involves several tasks and phases, as presented in this section.

[28] https://www.wikidata.org/wiki/Wikidata:Main_Page
[29] https://www.w3.org/TR/sparql11-query/
[30] https://www.w3.org/TR/sparql11-federated-query/

FAIR data principles

A KG is only one system within an enterprise environment consisting of interconnected systems. One key aspect in this mixture of interconnected systems and data is to follow the FAIR Data principles[31]:

- **Findable**: Metadata and data should be easy to find for both humans and computers. Machine-readable metadata is essential for automatic discovery of datasets and services.
- **Accessible**: Using data requires access documentation, possibly including authentication and authorization.
- **Interoperable**: Data usually needs integration with other data and use within applications or workflows for analysis, storage, and processing.
- **Reusable**: Metadata and data need good descriptions for replication and combination in different settings for this ultimate goal.

Iterative creation

It is often useful to apply an iterative approach to creating a KG from scratch in several steps:

- **Define data model** using ontologies and vocabularies.
- **Define RDF mappings** to convert from structured source data to RDF.
- **Preprocess source data** (per file) (e.g., clean-up data).
- **Perform RDF conversion** using the provided mappings.
- **Postprocess intermediate results** (per file) (e.g., create additional relations, aggregate data).
- **Load RDF data** into the KG to persist it in a graph database.
- **Postprocess intermediate results** (entire graph) (e.g., create additional relations, aggregate data).
- **Perform data validation** to ensure conformance to the defined data model.
- **Identify source datasets** and make them accessible.

Observing violations during data validation can be a starting point to improve the pipeline by fixing the source data through data cleaning, adjusting the ontology or RDF mappings, performing another iteration of the data integration process or ETL pipeline, and more.

Dataset metadata provision

Data catalogues are a core building block for any FAIR data implementation by connecting data assets with the KG. In this approach, the data catalogue is a KG semantically annotated with descriptive metadata and access metadata, interlinked with other parts of the KG such as ontologies and vocabularies, and

[31] https://www.go-fair.org/fair-principles/

embedded into and connected with data assets. Thus, it provides a unified access layer for the end user, with traceable and explainable connections down to original data sources. Dataset descriptions (or data catalogues) use open and extensible World Wide Web Consortium (W3C)[32] standards (e.g., DCAT[33]) to make the data discoverable, accessible, and traceable.

Metadata may provide information on several important aspects of a dataset:

- **Provenance** information to describe the data origin.
- **Lineage** information to record previously applied processing steps and data passing from one step in a mesh to another.
- **Licensing** information to specify the data's use, change or publishing.
- **Timestamp** of creation and last update to help understand the actuality.
- **Links** to dataset distributions for automated access and format selection.

With dataset descriptions, humans and machines (artificial intelligence (AI) algorithms) can consume data in context, directly linked to the models and descriptions based on open standards, shareable, and queried through a single semantic language.

4.2.6 Data Model Definition

A KG typically follows a well-defined schema (or multiple) specified using ontologies and vocabularies.

Ontologies

Ontologies are semantic data models that define the types of entities in a domain and their properties. An ontology combines a representation, formal naming, and elements defining the domain of discourse, such as classes, attributes, and relations. One can see it as a logical graph model that defines existing types (sets) of entities, their shared attributes, and logical relations. Open standards like web ontology language (OWL)[34,35] and shapes constraint language (SHACL)[36] specify ontologies.

Vocabularies

Vocabularies are controlled term collections organised in concept schemes that support KG experts, domain experts, and business users in capturing relevant terminology, like synonyms, hyponyms, spelling, and language variations. A term could include preferred and alternative labels (synonyms) in multiple languages and carries natural language definitions. Terms relate to each other, for example, equal, broader, narrower (hyponyms), or loosely related. Common

[32] https://www.w3.org/
[33] https://www.w3.org/TR/vocab-dcat-2/
[34] https://www.w3.org/OWL/
[35] https://www.w3.org/TR/owl-overview/
[36] https://www.w3.org/TR/shacl/

examples of vocabulary types are thesauri, taxonomies, terminologies, glossaries, classification schemes, and subject headings, managed using the open simple knowledge organization system (SKOS)[37,38] standard.

Multiple data model uses

A data model defined using ontologies and vocabularies has multiple purposes:

- **Documentation**: Ideally, including a graphical view of the data model in the documentation helps understand the data and find connections between entities.
- **Data mappings**: The source data model (e.g., a JSON file or schema of a relational database) needs mapping to the RDF data model for the conversion process. Identifying properties, relations or connections between entity types facilitates the authoring of mappings.
- **Data validation**: Defining the ontology using OWL and SHACL can automatically validate the database and ensure that data follows the model.
- **Explorative user interface**: Generic dataset exploration is much easier with fully described data. A KG or application engineer may build a custom user interface for the dataset, facilitated by good data model documentation.

4.2.7 RDF Mappings

The mapping process automatically enables the conversion of a huge amount of source data to RDF. Converting structured data can map certain elements and attributes from the source files to RDF using a set of mapping rules. For example, all values of a column in a CSV file or a table in a relational database can map to RDF statements with the row's unique key mapped to a subject IRI, the column to a predicate and the row value to the object position of a triple. Mapping rules can be either declarative or programmatic.

Declarative mappings

Declarative mappings follow the no-code approach, defined using a simple text editor or visual tools without requiring special programming skills. The mappings defined using the standardized relational mapping language (RML)[39] rely on RDF, such that the data model (ontology), mappings (RML maps), and instance data all use the same format. RML supports tabular (relational) and hierarchical data structures in CSV, JSON, or XML formats. Support for other formats is typically possible. While RML defines the mapping language, multiple mapping engine implementations (most open source) are available, usable as stand-alone tools or embedded into custom applications as a library.

[37] https://www.w3.org/2004/02/skos/
[38] https://www.w3.org/TR/skos-reference/
[39] https://rml.io/specs/rml/

Programmatic mappings

Implementing the mapping process using a custom program is the most flexible way to convert data to RDF. All means provided by the programming language and its ecosystem, such as frameworks and libraries, can access data in various formats. Also, language-specific connectors, such as Java database connectivity (JDBC), to access relational databases in the Java programming language or web service connectors provide great flexibility. The biggest advantage is full control over the mapping process, as any algorithm, data generation, use of caches and memory, navigating data structure, or control flow is possible.

Hybrid mapping

Using declarative mappings based on RML is the quickest and easiest way to implement mappings from structured data to RDF, as it follows a predefined approach which covers many UCs and formats but does not require special programming skills. A custom program is necessary when declarative mappings do not suffice for the mapping. While programmatic mapping allows for greater flexibility, it requires more effort and programming skills from people implementing a graph processing workflow. A hybrid mapping may be suitable whenever declarative mappings support most data structures with few complicated exceptions handled by custom coding.

4.2.8 Graph Storage

While processing graph data in memory is possible for isolated UCs or as an intermediate representation of transitive results, most UCs require KGs submitted and built as a persistent structure in a persistent storage medium. The storage may be as simple as a set of RDF files or a triple store in a graph database.

RDF files

Graph data stored in files is the most basic form of graph storage, allowing easy handling, versioning, or sharing of datasets by simply moving or processing files using established tool chains and distribution methods.

Various RDF serialization formats exist, such as Turtle (`.ttl`) and Turtle-star (`.ttls`), TRiG (`.trig`) and TriG-star (`.trigs`), JSON-LD (`.jsonld`), OWL/RDF/XML (`.xml`, `.rdf`, `.owl`), n-triples (`.n3`), n-quads (`.nq`) and a few more.

Besides using different syntax from established structured file formats (XML, JSON) or custom ones (Turtle, TRiG), the serialization formats also differ in their support for specifying contexts (named graphs) or support for RDF-star. Some formats (XML) are more verbose, while others are rather compact (Turtle, TRiG). RDF files are usually plain text files. To save storage space, they may be compressed, for example, in the zipped Turtle format (`.ttl.gz`).

Turtle(-Star) for basic graph data and TRiG(-Star) are the most commonly used formats for preserving named graph information.

Moving data between graph databases for stand-alone distribution of graph data uses RDF file formats.

RDF databases

RDF databases provide an optimized storage solution when working with large graph data. One can query the data stored using native graph storage following the RDF(-star) data model using the SPARQL query language.

Additionally, databases typically provide native service integrations for specific UCs, such as search indexes for full-text search based on Lucene or Elasticsearch or embeddings in a vector database for analytics and ML.

The RDF(-star) data model,[40,41] the SPARQL query language,[42] as well as the SPARQL[43] and graph-store[44] endpoints for easy access to the database are W3C standards for interoperability and portability.

Multiple open-source or free database engines and a selection of well-supported enterprise-grade commercial databases, such as GraphDB,[45] AWS Neptune,[46] and Stardog,[47] are available.

4.2.9 Pre- And Postprocessing

Additional steps are sometimes necessary to conform to the graph data model, performed as pre- or postprocessing steps on the original source data or after the RDF conversion.

Preprocessing steps typically apply to a single source file. Typical examples are data cleansing, invalid data filtering, splitting out units from numerical values, or datatype conversions to conform with certain numeric or date-time formats.

Postprocessing steps may apply to the intermediate RDF files or the entire graph. Typical examples are to specify the named graph for a set of statements, update graph metadata, such as the timestamp of the last update of a dataset based on source data, create links between entities, generate a SKOS vocabulary from keywords stored as attributes on some types of entities, rewrite or replace subject IRIs based on existing data or aggregate data.

4.2.10 Data Validation

Validation ensures good data quality after converting it to RDF and ingesting it in the database.

When defining the ontology using OWL and SHACL, the model description can automatically validate the database and ensure that data follows the data model. A so-called SHACL engine provided by (commercial) RDF databases, open

[40] https://www.w3.org/TR/rdf11-concepts/
[41] https://www.w3.org/2021/12/rdf-star.html
[42] https://www.w3.org/TR/sparql11-query/
[43] https://www.w3.org/TR/sparql11-protocol/
[44] https://www.w3.org/TR/sparql11-http-rdf-update/
[45] https://graphdb.ontotext.com/
[46] https://aws.amazon.com/neptune/
[47] https://www.stardog.com/platform/

source projects, or commercial tools such as metahistory verifies that the data in the database adheres to the shapes defined in the ontology.

Besides verifying the data conformance to the data model, additional checks include the existence of dataset metadata for each named graph or an expected set of named graphs after full ingestion to detect missing data.

Violations uncovered by the data validation can improve the pipeline by performing further data cleansing, adjusting the ontology or RDF mappings and performing another iteration of the data integration or ETL pipeline.

4.2.11 Graph Updates

After creation, keeping a KG up-to-date to reflect changes in the updated source datasets is fundamental and performed using different approaches.

Full update

The easiest approach is to recreate the whole graph whenever new or updated source data is available, following the original graph's creation process, with the new version replacing the previous one.

Interactive use may require creating a new graph version while the previous graph is still in use and swapping or replacing it once the creation is complete.

Incremental update

Recreating the full graph for any change might be costly, take a long time, or not feasible for big graphs or frequent but small changes to the source datasets. Incremental updates may help to isolate changes to the source datasets and apply them selectively to KGs.

Incremental updates require extracting newly added data as files converted to RDF and ingested into the database. This process needs to be aware of the added data partitioning strategy by placing the updated data in a new partition (i.e., named graph) with corresponding metadata (including updated timestamps) or simply by adding it to existing named graphs (e.g., for entity type). Changes in source data translate to deleting and adding statements in the corresponding RDF dataset, so careful tracking of obsolete data is necessary.

Applying incremental graph updates is complex and highly domain and data-source-specific, with great benefits to keeping the graph up-to-date without requiring a full recreation of the entire KG.

4.2.12 Automation

KG creation is not a one-time effort but repeated to keep the graph current and integrate new data sources. Automating this process is key to quick and efficient graph creation with repeatable outcomes and avoiding errors caused by manual execution of the required steps. An ETL pipeline, including a workflow

engine integrated with all relevant systems and services, can automate this process. The workflow must be resilient to transient errors in a distributed system landscape and customizable and adjustable to the user's runtime environment.

4.2.13 Runtime Deployment

The ETL pipeline needs IT infrastructure deployment consisting of servers and storage systems in three environments.

- **On-premise** hosted by the user's organisation provides great flexibility attenuated by the operational overhead to run and manage the infrastructure.
- **Public cloud** allows clever reuse of services for data storage, databases or serverless processing, thereby greatly reducing operational overhead while preserving some flexibility and integration with existing infrastructure and other deployments running in the same environment.
- **Software-as-a-Service (SaaS)** provides a ready-to-use pipeline as a managed service, taking care of all aspects of running the workflow upon providing the source data, RDF mappings and some parameters. SaaS does not offer much customization or flexibility but allows quick turnaround times concerning setup and getting results. ETL pipelines are not yet available from the main public cloud providers as ready-to-run SaaS offerings and might require specialized UCs from smaller innovative companies.

Section 5.2 provides further details about the deployment of the ETL pipeline.

4.3 WORKFLOW DEFINITION AND DEPLOYMENT

Figure 18 shows the UC-0 flow diagram across the Graph-Massivizer tools annotated with the (1) KG creation and (2) workflow definition and deployment according to the graph inception architecture depicted in Figure 17.

A graph processing workflow consists of BGOs realized by:

- Graph-Inceptor as a Java library;
- Graph-Scrutinizer as library functions and SaaS API.

In both cases, a **registry** of available BGOs and their descriptions will be available to the Graph-Optimizer.

The **computing environment** in Graph-Massivizer comprises two types of infrastructures considered for deployment.

- **On-premise HPC** infrastructure typically requires a workload management system such as **Slurm**[48] that allows users to schedule and run BGOs as a software program on computing nodes.
- **Cloud** infrastructure simplifies the deployment and execution of services using container technologies such as **Docker**[49] and management middleware

[48] https://slurm.schedmd.com/overview.html
[49] https://www.docker.com/

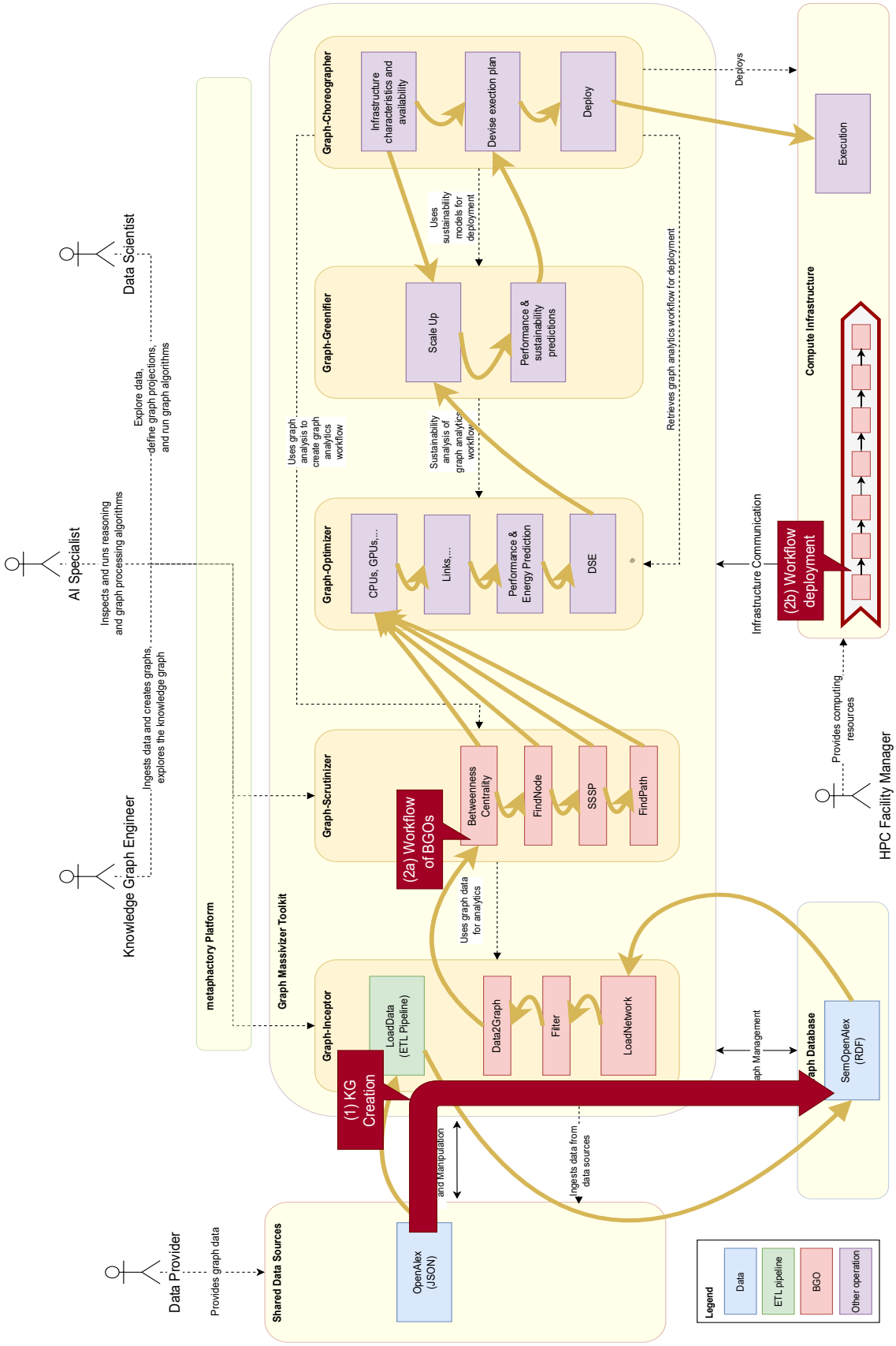


Figure 18: UC-0 flow across Graph-Massivizer tools (annotated).

like **Kubernetes** (K8s).[50] However, HPC systems are typically shared environments granting exclusive user access to data, while Docker gives superuser privileges, posing security challenges. **Singularity**[51] mitigates them by effectively executing in user mode interoperable with Docker images.

- **SaaS** allows provisioning BGOs deployed on cloud infrastructures as services and accessed from Slurms jobs on HPC clusters. Hybrid workflows with BGOs running on HPC clusters and clouds are possible.

From the stakeholder side, the tool (i.e., Graph-Inceptor, Graph-Scrutinizer) realizing the BGOs and the infrastructure (i.e., HPC or cloud) running them is irrelevant. However, the stakeholder must define the logical workflow execution in a tool-agnostic and infrastructure-agnostic manner automatically handled by the Graph-Massivizer platform. Thus, a workflow definition language understood and used by all tools in the Graph-Massivizer toolkit is necessary.

[50] https://kubernetes.io/
[51] https://docs.sylabs.io/guides/2.6/user-guide/singularity_and_docker.html

5 GRAPH-INCEPTOR DESIGN AND IMPLEMENTATION

This section describes the design and implementation of the initial preliminary release of Graph-Inceptor.

5.1 ARCHITECTURE DESIGN

Graph-Inceptor is a tool for creating KGs and processing and storing graph data. It offers two main services:

1. **KG creation pipeline** that:
   - supports various source data file formats;
   - uses declarative mappings (i.e., no programming required) to transform source data to graph data;
   - stores the resulting graph data in a graph database.
2. **Graph processing framework** that
   - reads source or graph data in various formats;
   - offers in-memory access and processing of graph data in various representations;
   - saves the resulting graph data in various graph file formats.

5.1.1 Implementation Components

Figure 19 shows a revised version of the Graph-Inceptor component architecture diagram presented in D2.1 (see Figure 24 in Annex B). The revisions consider two: (1) the work on defining and identifying BGOs as presented in this deliverable and (2) the development of two main services as separate components of the Graph-Inceptor tool.

Graph-Inceptor comprises two services: **graph creation** and **Graph Storage**. **Graph creation** component supports different types of graph processing workflows by a **KG Engineer**. As described above, Graph-Inceptor offers two main services. Two components implement these services:

1. **metaphacts ETL Pipeline**[52] provides a blueprint implementation for a *KG creation pipeline*. The pipeline can read various data sources, map them to RDF, and pre- and post-process the data. The result is a **(virtual) KG** stored in the graph storage. It is not necessary to convert all source data to RDF. A virtualization approach allows referencing original data in other databases, such as time series or binary files. Well-defined and standardised interfaces access and query the KG through SPARQL endpoints.
2. **Graphma**[53] is a graph processing framework that reads, assembles, and processes raw or graph data from different formats, offering in-memory access and integration across various processes, and saving the resulting graph

[52] https://github.com/graph-massivizer/graph-inceptor-etl-pipeline
[53] https://github.com/graph-massivizer/graph-inceptor-graphma

data in various graph file formats. Graphma supports a set of BGOs such as `LoadNetwork`, `Filter`, `Data2Graph`, `Interface2Graph`, `StoreGraph`, and `Graph2File`, identified in Table 3.

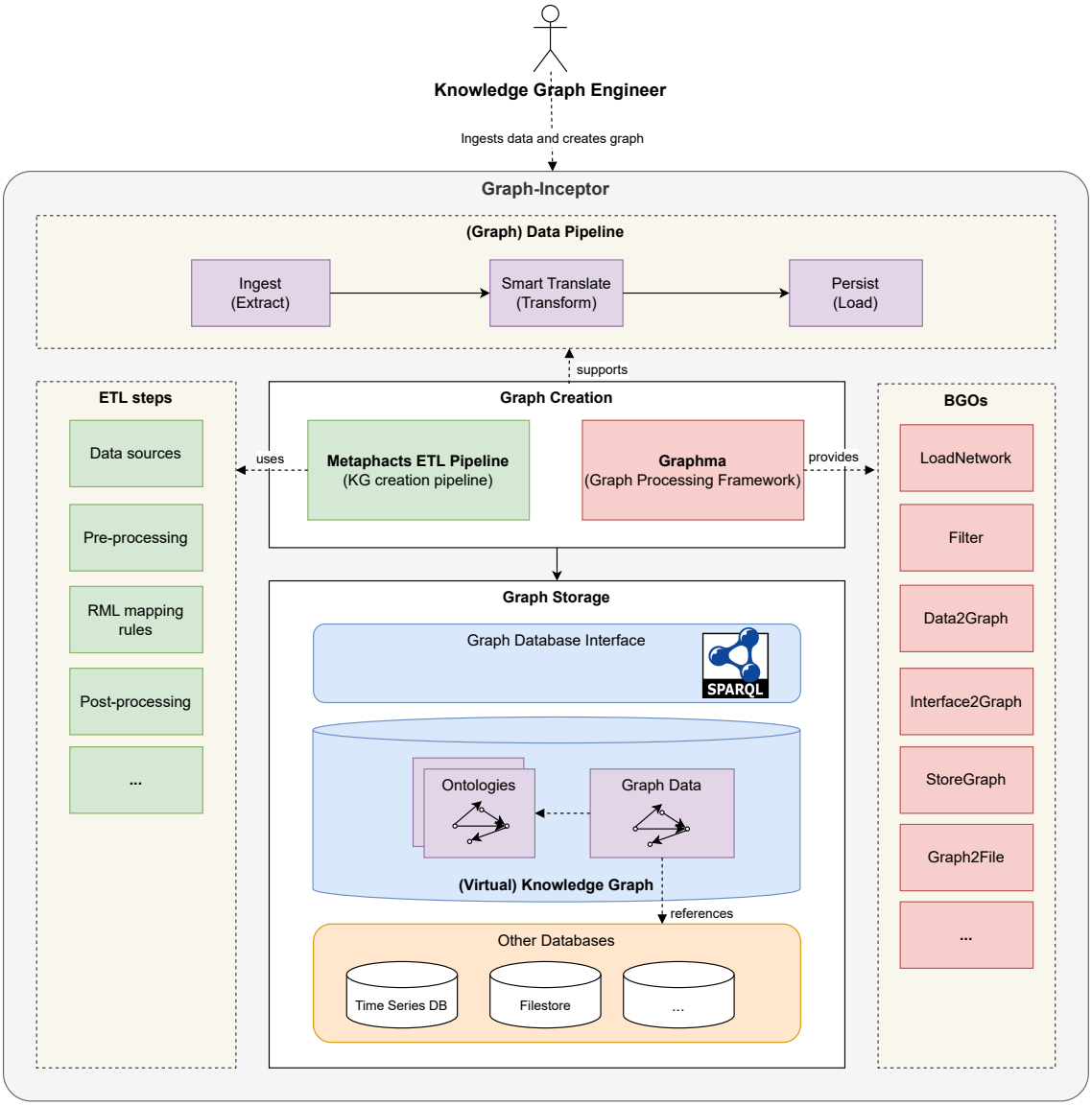


Figure 19: Graph-Inceptor component architecture diagram (revised).

5.1.2 Source Code Repositories

Graph-Inceptor has a single tool repository on GitHub used for documentation and integration tests (e.g., example data pipelines):

- `https://github.com/graph-massivizer/graph-inceptor`

The two Graph-Inceptor components have separate private implementation repositories that will become public during the project.

- `https://github.com/graph-massivizer/graph-inceptor-etl-pipeline`
- `https://github.com/graph-massivizer/graph-inceptor-graphma`

5.1.3 License

WP3 releases Graph-Inceptor open-source software under the Apache License, Version 2.0.[54]

5.1.4 Programming Languages

metaphacts ETL pipeline is a blueprint implementation on the AWS cloud that uses declarative mappings requiring no programming. However, one must define mappings in the RML language.

Graphma framework is a Java library currently supporting the Kotlin programming language. Presently, it considering using Apache Arrow to support other programming languages.

5.2 METAPHACTS ETL PIPELINE

The ETL pipeline provides a means to convert structured data to RDF, perform post-processing steps, and ingest it into a graph database, following the principles described in Section 4.2 using several components and tools:

- **AWS**[55] as a cloud environment with a selection of services for various parts
- **RML**[56] as declarative mapping language with Carml[57] as mapping engine;
- **Ontotext GraphDB**[58] as an RDF database.

5.2.1 Functionality

The ETL pipeline has the following functionality:

- Read source files from a simple storage service (S3) bucket;
- Convert source files to RDF using RML mappings;
- Support formats such as CSV, XML, JSON, JSONL, and compressed (ZIP);
- Write RDF files to an S3 bucket (one per source file);
- Ingest RDF files into a graph database using the GraphDB Preload[59] tool
- Add incremental file updates into the source bucket after the initial ingestion.

5.2.2 Architecture Design

Figure 20 shows the architecture diagram of the metaphacts ETL pipeline. The pipeline uses multiple cloud-native AWS services to implement the RDF conversion and ingestion, resulting in high parallelization and efficient resource use.

[54] https://www.apache.org/licenses/LICENSE-2.0
[55] https://aws.amazon.com/
[56] https://rml.io/specs/rml/
[57] https://github.com/carml/carml
[58] https://graphdb.ontotext.com/
[59] https://graphdb.ontotext.com/documentation/10.4/loading-data-using-importrdf.html#load-vs-preload

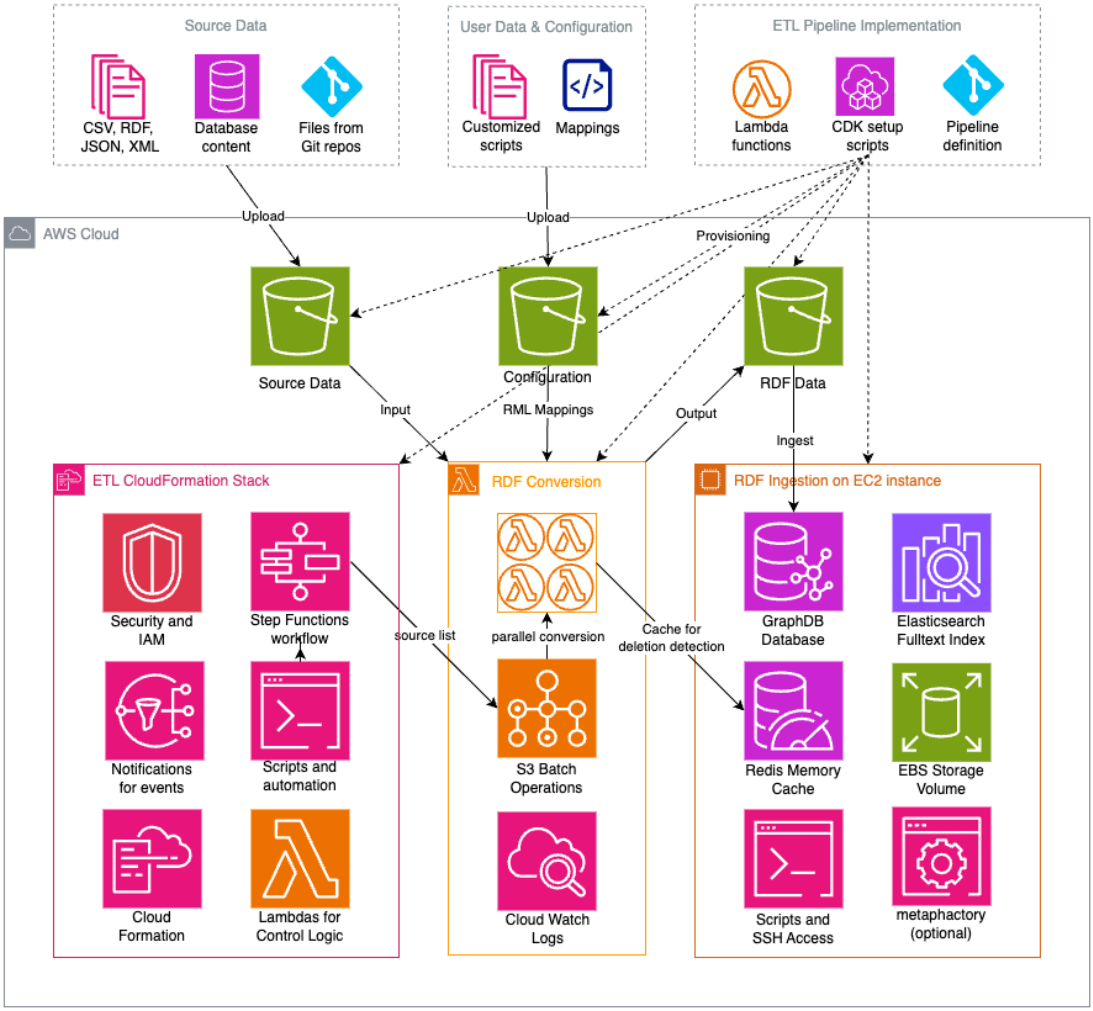


Figure 20: metaphacts ETL pipeline of Graph-Inceptor.

The pipeline uses the following services:

- **Cloud Formation**[60] and **the cloud development kit (CDK)**[61] to set up the pipeline and provision all resources;
- **S3**[62] to store input, configuration, output files and perform S3 Batch Operations[63] to convert source data to RDF;
- **Lambda**[64] for RDF conversion and various control logic;
- **Step functions**[65] to orchestrate the overall pipeline operational workflow, as described in Section 5.2.4.
- **Elastic Compute Cloud (EC2)**[66] **with elastic block storage (EBS)**[67] in-

[60] https://aws.amazon.com/cloudformation/
[61] https://aws.amazon.com/cdk/
[62] https://aws.amazon.com/pm/serv-s3/
[63] https://aws.amazon.com/s3/features/batch-operations/
[64] https://aws.amazon.com/pm/lambda/
[65] https://aws.amazon.com/step-functions/
[66] https://aws.amazon.com/pm/ec2/
[67] https://aws.amazon.com/ebs/

gest the RDF data into the RDF database including a full-text search index;

- **Simple notification service (SNS)**[68] for pipeline events to listeners (e.g., email address);
- **Identity and access management**[69] to create policies and roles and grant the permissions required to access services and resources.

An S3 bucket stores the configuration, user RML mappings and source files to convert and ingest for easy access from all AWS services. Another S3 bucket stores the output of the RDF conversion for keeping the RDF version of the source data or distributing it independently from the data in the RDF database.

5.2.3 Pipeline Creation

The pipeline follows the materialization approach described in Section 4.2.4, used whenever creating a KG from highly connected and structured source data, converted to RDF.

When integrating data from databases or other streaming sources, it needs to export and partition the relevant records into manageable pieces (e.g., files) upfront. This step keeps the high domain- and source-specific export and slicing of source data outside the pipeline difficult to implement generically. An example of such a step is exporting a big table from a relational database to a set of files with a fixed number of records per file.

In certain cases, data requires frequent updates from the original data sources, not ingested in one operation. The ETL pipeline supports this operation by either re-creating the graph from scratch for a new generation or updating it to apply incremental changes.

Creating a graph from scratch is the safest approach but might impose performance overheads for re-running the full RDF conversion and ingestion, even for already processed data. This works well for frequently changing data or complex data models, which are difficult to update in place.

Applying incremental updates highly depends on the domain model, which must identify and replace the changing parts. Due to the nature of the RDF data model, this operation requires deleting obsolete data before adding new information, which works well for infrequent or low-volume changes or for data that is easy to identify, such as a whole-named graph or outgoing edges of an entity.

5.2.4 Pipeline Workflow

The RDF conversion and ingestion process uses an AWS functions workflow that consists of several steps that orchestrate the involved services and applies scripting for some tasks running on the companion EC2 virtual machine instance. Some steps require additional control logic, such as selecting source files based on criteria like naming patterns or a timestamp, handled by additional lambda functions and allowing customization by overriding the provided

[68] https://aws.amazon.com/sns/
[69] https://aws.amazon.com/iam/

default implementations.

The high-level workflow consists of the following steps:

- **Source file selection** for processing;
- **Manifest creation** passing the source files to S3 batch operations for RDF conversion.
- **S3 batch operations** converting the source files to RDF using up to thousands of parallel AWS lambda functions for RDF conversion, running up to a few hours and retrying failed conversions to compensate for spurious errors;
- **Lambda conversion** of a source file to RDF data as follows:
  - Read the mapping configuration and all referenced mapping files from the S3 bucket;
  - Receive a list of source files to process from S3 batch operations;
  - Read each source file from the S3 bucket containing source files;
  - Determine the RML mappings applied by matching the file name to a list of mapping definitions, including additional pre- or post-processing hints.
  - Convert the source file to RDF, supporting source formats such as CSV, XML, JSON, JSONL, and compressed (gzipped);
  - Write RDF files to an S3 bucket (one RDF per source file).
- **SNS notification** after converting all source files;
- **Ingestion** on the companion EC2 instance as follows:
  - Prepare the ingestion by creating an empty database deployment;
  - Store all data on an RDF database using a flexible, detachable, movable, archivable, or clonable EBS volume;
  - Download the converted RDF files from the S3 bucket;
  - Launch the GraphDB Preload[70] tool in offline mode to ingest RDF files;
  - Send SNS notification after completing the ingestion;
  - Launch the GraphDB database;
  - Launch additional services like Elasticsearch and metaphactory (optional);
  - Create a full-text search index;
  - Send SNS notification after index creation;
- **Regular database operations**;
- **Incremental updating**, periodically checking for new data ingested into the running database, as follows:
  - Send notification using SNS;
  - Identify new files to ingest;

[70] https://graphdb.ontotext.com/documentation/10.4/loading-data-using-importrdf.html#load-vs-preload

- Perform conversion to RDF;
- Identify (domain-specific) changes from the original data set by naively inserting all data into the database or by detecting and deleting obsolete data using customized scripts in the S3 bucket holding the configuration;
- Delete obsolete data while running the database;
- Insert new or changed data;
- Sleep until the next iteration.

5.2.5 Setup And Operation

To set up and deploy an ETL pipeline, one needs to clone or download the definition, setup and helper scripts and lambda function implementations residing in the Git repository (see Section 5.1.2). Starting the setup requires the following pre-requisites AWS account,[71] AWS command line interface,[72] AWS CDK[73] for TypeScript, and Docker[74] for running and building local images.

The AWS CDK deploys all infrastructure components described in the architecture diagram in Section 5.2.2.by synthesizing CloudFormation descriptions of the infrastructure stack from the provided pipeline definition and additional configuration information provided by the user. Configuration parameters include an SSH key to access the provisioned EC2 virtual machine instance and the instance type specifying its available CPU and memory resources.

Besides setting up infrastructure components, the pipeline definition also creates all required roles and policies to grant the permissions required to run and access the various services and resources created during deployment. For security reasons, these policies are available for user approval before starting the actual deployment.

The documentation[75] in the Git repository describes the steps to set up the pipeline and the required configuration parameters in detail. Upload the source data and RML mappings manually to the respective S3 bucket or copy them in the respective folder and then automatically will be copied to the right place by triggering the deployment. See section 5.2.6 for an example of source data and RML mappings, to verify proper pipeline operation and as a starting point to develop the RML mappings for the intended UC.

5.2.6 Pipeline Example

This section gives an end-to-end example demonstrate the ETL pipeline used to verify proper execution and explore features and operations.

The Dimensions Covid Dataset[76] provides information on publications (academic papers), research grants, datasets, and clinical trials, available for down-

[71] https://repost.aws/knowledge-center/create-and-activate-aws-account
[72] https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html
[73] https://docs.aws.amazon.com/cdk/v2/guide/getting_started.html
[74] https://docs.docker.com/get-docker/
[75] https://github.com/graph-massivizer/graph-inceptor-etl-pipeline/blob/main/docs/Setup.md
[76] https://www.dimensions.ai/covid19/

loaded from Figshare[77] (approximately 1 GB). Deploying this dataset and run the pipeline requires the following steps:

- Set up the ETL pipeline, as described in Section 5.2.5.
- Copy the RML mappings from the example (`mappings`) folder into the main mappings folder in the root directory of the pipeline definition or directly into the S3 bucket containing the mappings using CDK deployment;
- Download and extract the source files from Figshare into the `source-files` folder or the source data S3 bucket;
- Make four CSV files available directly in the source folder, with the names matching those listed in `examples/dimensions-covid/dimensions-covid-request.json`;
- Copy the static RDF files (ontology, dataset description) from the example `source-files` folder into the `source-files` folder in the root directory of the pipeline definition or the source data S3 bucket using CDK deployment;
- Trigger the ETL workflow from the AWS step functions console[78] or wait for the predefined time-based trigger to start the workflow.

5.3 GRAPHMA

Graphma is a component of the Graph-Inceptor tool for graph generation and stream processing built upon the Magma core library. Magma provides the fundamental building blocks for creating stream pipelines that operate on a finite amount of data. Magma takes a "correct-by-construction" approach that leverages generic functional programming structures to compose rich types that encourage the participation of the Java-type checker. Graphma's foundation in Magma's "correct-by-construction" philosophy ensures reliability and efficiency. This approach ensures that every component of Graphma adheres to foundational principles, promoting error-free operation. This section introduces the Magma core functionalities and describes the status of the Graphma implementation at the time of writing, continuously improving during the next year and following the software release milestones of the project. Finally, it describes the planned features in Graphma component, including BGOs.

5.3.1 Magma Core Library

Pipeline

A pipeline described in the **Magma Compute Pipeline** package is a sophisticated tool for modelling and executing computations in a structured and efficient manner, designed to handle complex data processing tasks by organizing them into steps or stages. The pipeline is **heterogeneously-typed**, handling different data types safely and controlled. Its structured nature ensures consistent management of the data types moved through each pipeline stage. Cen-

[77] https://dimensions.figshare.com/articles/dataset/Dimensions_COVID-19_publications_datasets_and_clinical_trials/11961063

[78] https://us-east-1.console.aws.amazon.com/states/home?region=us-east-1#/statemachines

tral to the pipeline design is the concept of **laziness** or inertia. Instead of computing data immediately, it builds an execution plan outlining how data should be transformed through each pipeline stage, deferring actual computations until necessary and enhancing efficiency. The pipeline stages are:

- **Source** representing the beginning of the pipeline, entering external data;
- **Stages**, representing intermediate steps processing and transforming data, linked together in a chain of operations;
- **Sink**, representing the end of the pipeline and producing the final result.

The pipeline has different evaluation strategies:

- **Eager evaluation**, immediately executing the computations;
- **Lazy evaluation** deferring and executing computation only when needed.
- **Memorized evaluation**, performing computations once, and storing the results for quick access and reuse in future requests.

The pipeline offers the following features:

- **Flow control** over the flow of computation, allowing for optimizations in execution speed and memory use;
- **Traversal and transformation** through computed values moved through the various stages.
- **Composable operations** to achieve the desired data processing goals in different configurations.
- **Evaluation and execution** in a **driver** state machine that manages it according to the selected strategy.

In summary, pipelines are a powerful and flexible framework for structuring and executing complex data processing tasks. They emphasize type safety, lazy computation, and flexible evaluation strategies to optimize performance and resource utilization. This approach makes it suitable for a wide range of data-intensive applications. Magma provides the pipeline abstraction and a set of basic operators divided into three categories.

Source/traverse operators

A **Traverser**, outlined in the **Magma Control Traversal** package, is a tool for navigating through a collection of items in an organized way. It is a specialised guide and smart cursor for visiting each item in a list or array of numbers, characters, or complex data structures. The Traverser allows for two navigation types:

1. **Step-by-step movement** iterates each item separately until the end. **Duplex Traverser** is an advanced version that allows moving forwards and backwards through the collection;
2. **Bulk processing** traverses multiple items in parallel and applies a specific action to each until processing all items or meeting a certain condition.

A Traverser adaptation for standard Java programming structures represents a versatile tool for many programming scenarios, useful for integrating existing code or systems.

Intermediate operators

Intermediate operators lazily transform pipes (the building blocks of pipelines):

- **Stateless intermediate operators** require only a very small amount of state to operate and retain no state from previous values when independently processing new ones.;
- **Stateful intermediate operators** may incorporate states from previous values when processing new ones and require a significant amount of state, usually proportional to the number of values in a sequence;
- **Blocking operators** process the entire input before producing output (e.g., sort in a Java stream);
- **Pipelining operators** process the entire input before producing output.

Magma core library implements several intermediate operators documented in the Magma codebase under `magma-core/src/java/magma/compute`: `Append`, `Difference`, `Distinct`, `Drop`, `Filter`, `FlatMap`, `Front`, `GroupBy`, `Insert`, `Intersect`, `Intersperse`, `Join`, `Map`, `Peek`, `Prepend`, `Replace`, `Reverse`, `Slice`, `Sort`, `Tail`, `Take`, `Union`, and `Zip`.

Terminal operators

Terminal operators are the concluding operations in a pipeline that yield a result or trigger a side effect. Their activation initiates the pipeline assembly phase that visits all operators within its description from bottom to top, weaving them into one another and forming the requisite contexts. The end product is an instantiated pipeline.

This assembly represents a transition from a logical to a physical execution plan. Executing a terminal operation consumes the pipeline, which is unavailable for further operations. However, a flush and reset mechanism allows for its reusability depending on the pipeline type.

Magma core library implements the following terminal operators: `Aggregate`, `All`, `Any`, `At`, `Collect`, `Contains`, `Convert`, `Count`, `Find`, `First`, `Fold`, `ForNext`, `IndexOf`, `Last`, `Max`, `Min`, `None`, `Reduce`, and `Single`

5.3.2 Graphma Extensions

Graphma enhances the Magma core library pipeline implementation with custom operators tailored for graph processing. These new operators are mostly BGOs defined in Section 2.1.1, encapsulating functionalities from the JGraphT Java package [76]. Graphma's strategic vision involves a progressive transition to high-performance, custom solutions.

Compared to Magma, Graphma emphasises cross-language compatibility and flexibility. Graphma aims to integrate Apache Arrow, offering off-heap memory data structures shared across processes and enabling implementations in programming languages other than Java. This integration marks a significant step towards creating a more versatile and inclusive graph processing environment. It allows developers to leverage Graphma's capabilities in their preferred programming language, making it a more accessible and adaptable tool for various graph processing applications.

5.3.3 Repository Structure

The Graphma Java project facilitates efficient and community-oriented development focusing on data management. It leverages Gradle, configured with Kotlin for build domain-specific automation. Additionally, it uses a local Maven repository for managing dependencies. The Java module integrated with Gradle enhances modularity and project organization and incorporates the Magma core library in the local Maven repository via a custom directory .

Comprehensive unit testing is key to our development process, especially for the Magma core library. Section 5.3.1 list the tests for all lazy and strict operators, demonstrating commitment to thorough validation, robustness and reliability, including pipeline assembly, underlying buffers, and traversers.

Graphma implements unit tests covering the existing JGraphT BGO operator wrappings. Additionally, tests developed for the custom Parquet file parser further enhances the data processing capabilities.

A special testing feature is the SuiteSparse dataset included through a dedicated Gradle module designed as a singleton database for querying of graph sets with various properties. Magma's pipeline filters efficiently filter these properties, showcasing advanced data handling and testing capabilities.

6 GRAPH-SCRUTINIZER DESIGN AND IMPLEMENTATION

This section describes the Graph-Scrutinizer and provides insights into the current implementation.

6.1 ARCHITECTURE DESIGN

The Graph-Scrutinizer tool is a comprehensive set of graph processing and analytics capabilities. It encompasses graph distillation (e.g., graph sampling, summarization, partitioning), graph analytics (e.g., query answering, graph traversal), and graph ML (e.g., graph clustering, node classification). These capabilities are available for batch and streaming settings in accordance with the UCs. The tool is available in different programming languages as a library or service for particular graph processing implementations. A homogeneous interface will be available where possible and desired but will sacrifice homogeneity for efficiency in specific cases. The tool receives requests which specify the operation to apply on data (batch or streaming) and the location of the result. It then decomposes these requests into BGOs and forwards them with the available metadata of the implementations.

The Graph-Scrutinizer tool [77] realizes nearly twenty BGOs, described in Table 3, to enable scalable, low-latency, low-memory footprint graph processing in different formats based on UC requirements.

6.1.1 Implementation Components

The Graph-Scrutinizer consists of three main components shown in Figure 21.

Graph analytics

It is concerned with executing queries against graphs. The component supports batch and streaming analytics and standing queries.

- **Graph distillation** supports three strategies for dealing with graphs at scale:
  - **Graph sampling** obtains a graph of reduced size with certain nodes and edges;
  - **Graph partitioning** divides the graph into sub-graphs separate processing result aggregation;
  - **Graph summarization** represents large graphs in a structured and compact way, preserving relevant properties.
- **Graph algorithms and query** are of two kinds:
  - **Exact** using as SPARQL* queries and popular heuristics such ad breadth and depth-first searches;
  - **Approximate** using existential first-order logic, graph generators, sketch.

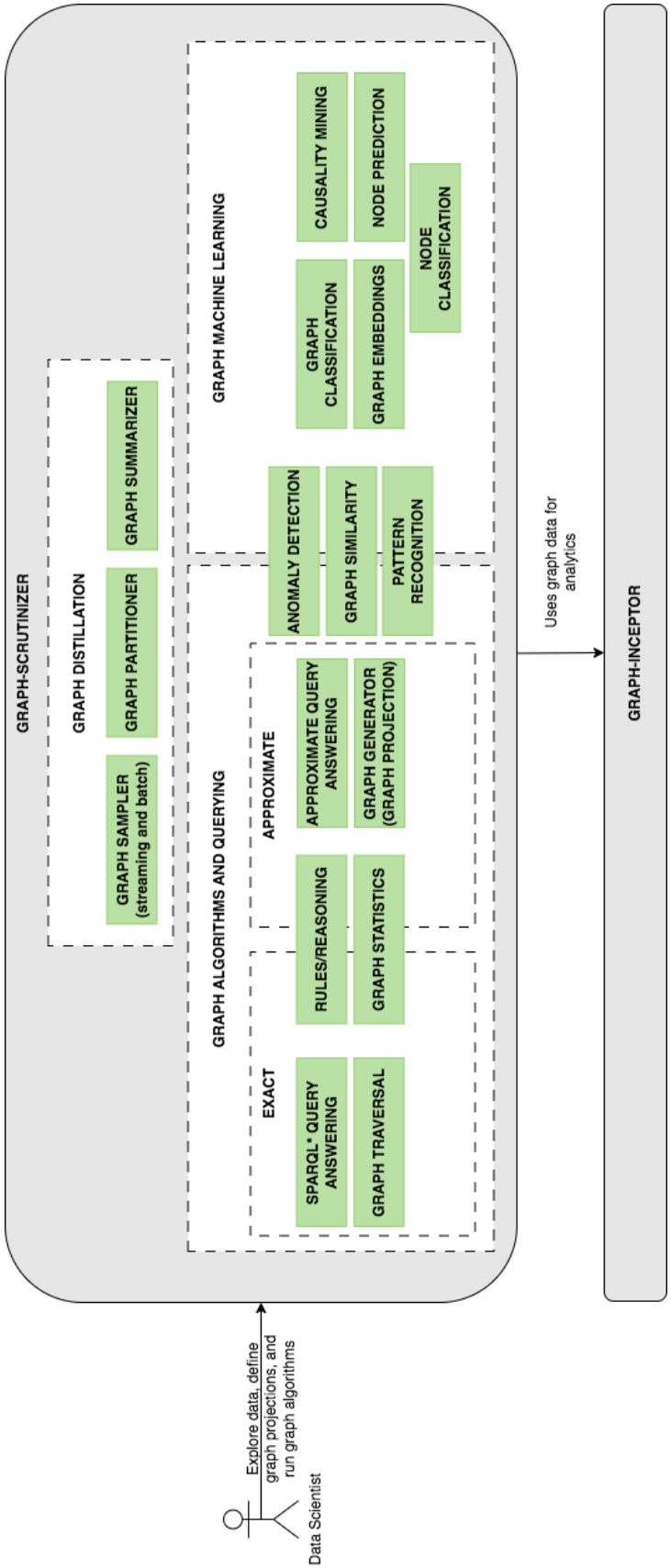


Figure 21: Graph-Scrutinizer, component diagram.

Finally, reasoning and graph statistics implementations can return exact or approximate results.

- **Graph ML** comprising graph clustering, graph embeddings, causality mining, node prediction and classification, anomaly detection, pattern recognition, and graph similarity methods.

6.2 REPOSITORY STRUCTURE

Graph-Scrutinizer adopts the trunk-based development methodology[79] coupled with continuous integration and deployment. The methodology requires a main branch ready for release and avoids other long-lived branches. Engineers must invest in extensive automated testing and code checks to ensure a code free of bugs and vulnerabilities. The testing strategy will consider at least using unit tests and monitoring multiple metrics of relevance (e.g., code coverage). The artefacts released follow the semantic versioning convention.

The single tool-level repository will provide documentation and integration test purposes, and a revised repository structure with multiple repositories, each having its own release cycle, will consider three artefacts:

- **Graph-Scrutinizer execution engine**, structured as a Python project, responsible for the BGO registry, receiving graph processing requests and converting them into a BGO workflow;
- **Library service wrapper**, also as a standard Python project, implementing streaming and representational state transfer (REST) APIs to gather BGO requests and request BGOs execution;
- **BGO implementations**, either as a standard Python project or following the best practices of other programming languages, if necessary.

[79] https://trunkbaseddevelopment.com/

7 CONCLUSIONS AND FUTURE WORK

This deliverable presented the results of the three tasks carried out in WP3 during the first year of the Graph-Massivizer project:

- **Task T3.1** on massive graph creation;
- **Task T3.2** on graph queries and ML at an extreme scale;
- **Task T3.4** on an interface for graph operators carried out in WP3 during the first year:

These tasks successfully achieved the following results:

- **Research overview and analysis** of BGOs, graph creation and synthetic generation approaches, graph benchmarks frameworks and tools, and distributed graph sampling techniques.
- **Definition of key BGO terms** coupled with **UC and platform requirements analysis** from the deliverable D2.1 to identify a set of BGOs to support.
- **Revised Graph-Massivizer architecture** from the deliverable D2.1, focusing on the Graph-Inceptor and Graph-Scrutinzer tools and their role in supporting BGOs with a baseline illustrative example called UC-0.
- **Preliminary releases of Graph-Inceptor and Graph-Scrutinizer** tools under the Apache 2 license in well-defined Gihub repositories.

Graph-Inceptor and Graph-Scrutinizer will continue their development during the next project phases of the project with official releases according to the overall project **milestones (MS)**:

- **MS-2: Preliminary Graph-Massivizer tool prototypes** targeted at M15 will result from two steps;
  1. **M12** concluded the first development iteration of the toolkit architecture design based on a detailed requirements analysis and released a preliminary version of the tools;
  2. **M15** with conduct validation campaigns with external beta-testers and focus group members and collect feedback using individual tools.
- **MS-3: First validated Graph-Massivizer toolkit** targeted at M27 will result from two steps;
  - **M24:** The first development iteration will incorporate stakeholder feedback from the first step, evolve the requirements, revise the toolkit architecture and release a first alpha-tested integrated toolkit.
  - **M27:** The second step will conduct validation campaigns with selected external beta-testers and focus group members and collect feedback using the integrated toolkit and the operational UCs.
- **MS-4: Final validated Graph-Massivizer toolkit** targeted at M36 repeats another development iteration on one step;

- **M36:** The final validation phase will evaluate the key performance indicators associated with the technical objectives and the UCs.

More frequent internal tool releases will make earlier features available. These achievements provide a strong foundation for the ongoing and future work in WP3, concluded with the final deliverable D3.2 on **"Massive graph enrichment, query, inference and BGO interfaces"** due in M33.

8 REFERENCES

[1] Turra, Roberta, et al. *D1.3: Graph-Massivizer Data Management Plan*. Graph-Massivizer, 2023, graph-massivizer.eu/.

[2] Dizaji, Haleh, et al. *D2.1: Graph-Massivizer Requirements Elicitation and First Architecture Design*. Graph-Massivizer, 2023, graph-massivizer.eu/.

[3] Eberhart, Aaron, et al. *D6.1: Graph-Massivizer infrastructure, tools and use cases*. Graph-Massivizer, 2023, graph-massivizer.eu/.

[4] Wang, Lei, and Minghe Yu. "GraphBench: A Benchmark Suite for Graph Computing Systems". *Benchmarking, Measuring, and Optimizing*. Ed. by Wanling Gao et al., Springer International Publishing, 2020, pp. 20–31.

[5] Leskovec, Jure, and Rok Sosič. "SNAP: A General-Purpose Network Analysis and Graph-Mining Library". *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, July 2016.

[6] Ni, Qingjian, and Meng Zhang. "STGMN: A gated multi-graph convolutional network framework for traffic flow prediction". *Applied Intelligence*, vol. 52, no. 13, 2022, pp. 15026–39.

[7] Färber, Michael, et al. "SemOpenAlex: The Scientific Landscape in 26 Billion RDF Triples". *The Semantic Web – ISWC 2023*. Ed. by Terry R. Payne et al., Springer Nature Switzerland, 2023, pp. 94–112.

[8] Hu, Weihua, et al. "Open graph benchmark: Datasets for machine learning on graphs". *Advances in neural information processing systems*, vol. 33, 2020, pp. 22118–33.

[9] Capotă, Mihai, et al. "Graphalytics: A big data benchmark for graph-processing platforms". *Proceedings of the GRADES'15*, 2015, pp. 1–6.

[10] Dayarathna, Miyuru, and Toyotaro Suzumura. "Benchmarking graph data management and processing systems: A survey". *arXiv:2005.12873*, 2020.

[11] Zhang, Fangyan, et al. "A visual and statistical benchmark for graph sampling methods". *Exploring Graphs at Scale Workshop*. 2015.

[12] Coscia, Michele, and Luca Rossi. "Benchmarking API costs of network sampling strategies". *2018 IEEE international conference on big data (Big Data)*. IEEE, 2018, pp. 663–72.

[13] Bader, David A, et al. "HPC scalable graph analysis benchmark". *Citeseer. Citeseer*, vol. 2009, 2009, pp. 1–10.

[14] Murphy, Richard C, et al. "Introducing the graph 500". *Cray Users Group (CUG)*, vol. 19, 2010, pp. 45–74.

[15] Dayarathna, Miyuru, and Toyotaro Suzumura. "Xgdbench: A benchmarking platform for graph stores in exascale clouds". *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 363–70.

[16] Ammar, Khaled, and M Tamer Özsu. "Wgb: Towards a universal graph benchmark". *Advancing Big Data Benchmarks: Proceedings of the 2013 Workshop Series on Big Data Benchmarking, WBDB. October 9-10, 2013*. Springer, 2014, pp. 58–72.

[17] Armstrong, Timothy G, et al. "Linkbench: a database benchmark based on the facebook social graph". *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 1185–96.

[18] Jouili, Salim, and Valentin Vansteenberghe. "An empirical comparison of graph databases". *2013 International Conference on Social Computing*. IEEE, 2013, pp. 708–15.

[19] Wang, Lei, et al. "Bigdatabench: A big data benchmark suite from internet services". *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*. IEEE, 2014, pp. 488–99.

[20] Zhu, Yuqing, et al. "Bigop: Generating comprehensive big data workloads as a benchmarking framework". *Database Systems for Advanced Applications: 19th International Conference, DASFAA 2014, April 21-24, 2014. Proceedings, Part II 19*. Springer, 2014, pp. 483–92.

[21] Xia, Fan, et al. "Bsma: A benchmark for analytical queries over social media data". *Proceedings of the VLDB Endowment*, vol. 7, no. 13, 2014, 1573–1576.

[22] Beamer, Scott, Krste Asanović, and David Patterson. "The GAP benchmark suite". *arXiv:1508.03619*, 2015.

[23] Nai, Lifeng, et al. "GraphBIG: understanding graph computing in the context of industrial solutions". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[24] Li, Min, et al. "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark". *Proceedings of the 12th ACM international conference on computing frontiers*. 2015, pp. 1–8.

[25] Iosup, Alexandru, et al. "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms". *Proceedings of the VLDB Endowment*, vol. 9, no. 13, 2016, pp. 1317–28.

[26] Dhulipala, Laxman, et al. "The graph based benchmark suite (GBBS)". *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 2020, pp. 1–8.

[27] Erb, Benjamin, et al. "Graphtides: a framework for evaluating stream-based graph processing platforms". *Proceedings of the 1st ACM SIGMOD joint international workshop on graph data management experiences & systems (GRADES) and network data analytics (NDA)*. 2018, pp. 1–10.

[28] Xu, Zhen, et al. "Gardenia: A graph processing benchmark suite for next-generation accelerators". *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 1, 2019, pp. 1–13.

[29] Zhang, Chao, et al. "Unibench: A benchmark for multi-model database management systems". *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, August 27–31, 2018*. Springer, 2019, pp. 7–23.

[30] Lim, Derek, et al. "Large scale learning on non-homophilous graphs: New benchmarks and strong simple methods". *Advances in Neural Information Processing Systems*, vol. 34, 2021, pp. 20887–902.

[31] Palowitch, John, et al. "Graphworld: Fake graphs bring real insights for gnns". *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 3691–701.

[32] Duan, Keyu, et al. "A comprehensive study on large-scale graph training: Benchmarking and rethinking". *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 5376–89.
[33] Huang, Shenyang, et al. "Temporal graph benchmark for machine learning on temporal graphs". *arXiv:2307.01026*, 2023.
[34] Fey, Matthias, et al. "Relational Deep Learning: Graph Representation Learning on Relational Databases". *arXiv:2312.04615*, 2023.
[35] Suzumura, Toyotaro, et al. "Performance characteristics of Graph500 on large-scale distributed environment". *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 2011, pp. 149–58.
[36] Iosup, Alexandru, et al. "The LDBC Graphalytics Benchmark". *CoRR*, vol. abs/2011.15028, 2020. *arXiv*, arxiv.org/abs/2011.15028.
[37] Iosup, Alexandru, et al. "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms". *Proceedings of the VLDB Endowment*, vol. 9, no. 13, 2016, pp. 1317–28.
[38] Dhulipala, Laxman, Guy E Blelloch, and Julian Shun. "Theoretically efficient parallel graph algorithms can be fast and scalable". *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 1, 2021, pp. 1–70.
[39] Hu, Weihua, et al. "Ogb-lsc: A large-scale challenge for machine learning on graphs". *arXiv:2103.09430*, 2021.
[40] Leskovec, Jure. "Kronecker Graphs (presentation)". 2008.
[41] Chakrabarti, Deepayan, Yiping Zhan, and Christos Faloutsos. "R-MAT: A recursive model for graph mining". *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–46.
[42] Lothian, Joshua, et al. "Synthetic graph generation for data-intensive HPC benchmarking: Background and framework". *Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2013/339*, 2013.
[43] Khorasani, Farzad, Rajiv Gupta, and Laxmi N. Bhuyan. "Scalable SIMD-Efficient Graph Processing on GPUs". *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*. PACT '15, 2015, pp. 39–50.
[44] Spasic, Mirko, Milos Jovanovik, and Arnau Prat-Pérez. "An RDF Dataset Generator for the Social Network Benchmark with Real-World Coherence." *BLINK@ ISWC*. 2016.
[45] Erling, Orri, et al. "The LDBC social network benchmark: Interactive workload". *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 619–30.
[46] Zhang, JW, and YC Tay. "GSCALER: Synthetically Scaling A Given Graph." *EDBT*. 2016, pp. 53–64.
[47] Gutfraind, Alexander, Ilya Safro, and Lauren Ancel Meyers. "Multiscale network generation". *2015 18th International Conference on Information Fusion (Fusion)*. IEEE, 2015, pp. 158–65.
[48] Staudt, Christian L, et al. "Generating scaled replicas of real-world complex networks". *International Workshop on Complex Networks and their Applications*. Springer, 2016, pp. 17–28.
[49] Musaafir, Ahmed, et al. "A Sampling-Based Tool for Scaling Graph Datasets". *ACM/SPEC ICPE'20*. ICPE '20, Association for Computing Machinery, 2020, pp. 289–300.

[50] Hubert, Nicolas, et al. PyGraft: Configurable Generation of Schemas and Knowledge Graphs at Your Fingertips. 2023. *arXiv*, arxiv.org/abs/2309.03685.
[51] Goyal, Nikhil, Harsh Vardhan Jain, and Sayan Ranu. "GraphGen: A Scalable Approach to Domain-Agnostic Labeled Graph Generation". *Proceedings of The Web Conference 2020*. WWW '20, Association for Computing Machinery, 2020, pp. 1253–63.
[52] Zhu, Yanqiao, et al. "A Survey on Deep Graph Generation: Methods and Applications". 2022.
[53] You, Jiaxuan, et al. "Graphrnn: Generating realistic graphs with deep auto-regressive models". *International conference on machine learning*. PMLR, 2018, pp. 5708–17.
[54] Silva, Vanessa Freitas, et al. "Time series analysis via network science: Concepts and algorithms". *WIREs Data Mining and Knowledge Discovery*, vol. 11, no. 3, 2021, e1404.
[55] Yin, Xingkun, et al. "Forecasting Stock Prices Using Stock Correlation Graph: A Graph Convolutional Network Approach". *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021.
[56] Leskovec, J., and C. Faloutsos. "Sampling from large graphs". *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.
[57] Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs". *Advances in neural information processing systems*, vol. 30, 2017.
[58] Chen, Jie, Tengfei Ma, and Cao Xiao. "Fastgcn: fast learning with graph convolutional networks via importance sampling". *arXiv:1801.10247*, 2018.
[59] Zou, Difan, et al. "Layer-dependent importance sampling for training deep and large graph convolutional networks". *Advances in Neural Information Processing Systems*, vol. 32, 2019.
[60] Huang, Wenbing, et al. "Adaptive Sampling Towards Fast Graph Representation Learning". *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al., Curran Associates, 2018.
[61] Yoon, Minji, et al. "Performance-Adaptive Sampling Strategy Towards Fast and Accurate Graph Neural Networks". *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21, Association for Computing Machinery, 2021, pp. 2046–56.
[62] Kipf, Thomas N, and Max Welling. "Semi-supervised classification with graph convolutional networks". *arXiv:1609.02907*, 2016.
[63] Zeng, Hanqing, et al. "Graphsaint: Graph sampling based inductive learning method". *arXiv:1907.04931*, 2019.
[64] Liu, Ziqi, et al. "Bandit samplers for training graph neural networks". *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 6878–88.
[65] Chiang, Wei-Lin, et al. "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks". *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 257–66.
[66] Wu, Felix, et al. "Simplifying graph convolutional networks". *International conference on machine learning*. PMLR, 2019, pp. 6861–71.

[67] Frasca, Fabrizio, et al. “Sign: Scalable inception graph neural networks”. *arXiv:2004.11198*, 2020.
[68] Sun, Chuxiong, Hongming Gu, and Jie Hu. “Scalable and adaptive graph neural networks with self-label-enhanced training”. *arXiv:2104.09376*, 2021.
[69] Mostafa, Hesham, et al. “FastSample: Accelerating Distributed Graph Neural Network Training for Billion-Scale Graphs”. *arXiv:2311.17847*, 2023.
[70] Zhu, Rong, et al. “AliGraph: A comprehensive graph neural network platform”. *arXiv:1902.08730*, 2019.
[71] Zhang, Fangyan, Song Zhang, and Christopher Lightsey. “Implementation and Evaluation of Distributed Graph Sampling Methods with Spark”. *Electronic Imaging*, vol. 30, no. 1, 2018, pp. 379-1–379-1.
[72] Jiang, Peng, and Masuma Akter Rumi. “Communication-efficient sampling for distributed training of graph convolutional networks”. *arXiv:2101.07706*, 2021.
[73] Tripathy, Alok, Katherine Yelick, and Aydin Buluc. “Distributed Matrix-Based Sampling for Graph Neural Network Training”. *arXiv:2311.02909*, 2023.
[74] Wang, Minjie, et al. “Deep graph library: A graph-centric, highly-performant package for graph neural networks”. *arXiv:1909.01315*, 2019.
[75] Nuseibeh, Bashar, and Steve Easterbrook. “Requirements Engineering: A Roadmap”. *Proceedings of the Conference on The Future of Software Engineering*. ICSE ’00, Association for Computing Machinery, 2000, pp. 35–46.
[76] Michail, Dimitrios, et al. “Jgrapht—a java library for graph data structures and algorithms”. *ACM Transactions on Mathematical Software (TOMS)*, vol. 46, no. 2, 2020, pp. 1–29.
[77] Rozanec, Joze, et al. “Graph-Scrutinizer: Towards Massive Graph Analytics and Reasoning”. *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 251–52.
[78] Rozanec, Joze, et al. “Graph-Inceptor: Towards Extreme Data Ingestion, Massive Graph Creation and Storage”. *Companion of the 2023 ACM / SPEC International Conference on Performance Engineering*. 2023, 253–254.
[79] Heymann, Sebastien, et al. “GEXF file format”. *GEXF Working Group,[online]. Available at WWW:< http://gexf. net/format*, 2009.
[80] Brandes, Ulrik, et al. Graph markup language (GraphML). 2013.
[81] Himsolt, Michael. *GML: A portable graph file format*. Technical report, Universitat Passau, 1997.

A USE CASE AND PLATFORM REQUIREMENTS

This section analyses the UC and platform requirements from the deliverable D2.1 that are relevant to the work in WP3. We retrieved the relevant UC and platform requirements from D2.1 and annotated them as follows:

- Yellow highlights the relevant text Graph-Inceptor and Graph-Scrutinizer technical requirements;
- Pink denotes revisions or corrections in the original requirements analysis of Graph-Inceptor and Graph-Scrutinizer ;
- Green denotes additional details added to the original Graph-Inceptor and Graph-Scrutinizer requirements analysis.

A.1 UC-1: GREEN AND SUSTAINABLE FINANCE

Table 5 : UC-1 Green and Sustainable Finance.

Functionality	Description	Tool
Financial data pre-processing and mapping	Ingest, preprocess and map historical financial data to a KG.	Graph-Inceptor
Financial patterns and correlations detection	• Find patterns and correlations across financial entities, concepts, and statistical features. • Identify anomalies, indicators, and recurrent behaviours.	Graph-Scrutinizer
Data interpolation	Interpolate missing data using inference and reasoning.	Graph-Scrutinizer
Synthetic financial graph generation	Generate synthetic financial graphs from extreme synthetic data.	Graph-Scrutinizer
Quality rules implementation	Implement quality rules and correlations, and detect relationships and patterns for synthetic data.	Graph-Scrutinizer
Synthetic financial data generation	• Generate synthetic data preserving the historical relationships, patterns, and statistical features. • Interpolate missing data, obeying the quality rules. • Integrate external third-party synthetic data generators for different financial domains.	Graph-Scrutinizer Graph-Optimizer Graph-Greenifier Graph-Choreographer
Green financial processing	• Estimate the impact of generating synthetic data. • Provide energy consumption scaling for all involved processes in synthetic data generation.	Graph-Greenifier

Functionality	Financial data preprocessing and mapping
Requirement identifier	UC-1.1.1
Requirement name	Financial data preprocessing
Priority	High
Description	Provide historical financial data in XML and CSV format to the Graph-Inceptor tool, which ingests and transforms it into the internal Graph-Massivizer format.
Precondition	Graph-Inceptor

Functionality	Financial data preprocessing and mapping
Requirement identifier	UC-1.1.2
Requirement name	Financial data mapping to a massive financial KG
Priority	Very high
Description	Map the historical financial data to a KG structure capturing all data quality rules.
Precondition	

Functionality	Financial patterns and correlations detection
Requirement identifier	UC-1.2
Requirement name	Relationships and patterns findings across financial entities
Priority	High
Description	• Find relationships and patterns across financial entities and concepts • Identify anomalies, potential indicators and recurrent behaviours
Precondition	Graph-Scrutinizer; UC-1.1.2

Functionality	Missing financial data interpolation
Requirement identifier	UC-1.3
Requirement name	Financial data interpolation
Priority	Medium
Description	Interpolate missing data using inference and reasoning.
Precondition	Graph-Scrutinizer; UC-1.1.2

Functionality	Synthetic financial graph generation
Requirement identifier	UC-1.4
Requirement name	Synthetic financial graph generation
Priority	High
Description	Generate synthetic financial data using GCNs and create inter-company and inter-product KGs.
Precondition	Graph-Scrutinizer; UC-1.2; UC-1.3

Functionality	Quality financial rules implementation
Requirement identifier	UC-1.5
Requirement name	Quality financial rules implementation
Priority	Very high
Description	Implement quality rules and correlations and detect relationships and patterns on the KG to generate synthetic financial data sets.
Precondition	Graph-Scrutinizer

A.2 UC-2: GLOBAL FORESIGHT FOR ENVIRONMENTAL PROTECTION

Table 6 : UC-2 Global Foresight for Environmental Protection.

Functionality	Description	Tool
Global news data ingestion	• Use open web sources and historical data, primarily global news data, to build a contextual KG • Encapsulate in KG entities and relationships between news metadata	Graph-Inceptor
Global news data modelling	• Build a KG of global foresight on environmental and social protection • Perform predictive analysis scenarios using Graph-Scrutinizer • Feed the model with hypothetical changes (e.g., a new line of events), and propagate this change through the graph using GNN, yielding predictions on the potential edges and nodes	Graph-Inceptor Graph-Scrutinizer
Global news data visualization	• Provide an easy-to-use frontend tool for a comprehensive view of future trends and scenarios to inform policy-making decisions better • Enable interpretability of GNNs and identify relevant graph parts contributing to the model predictions • Understand how and why certain environmental or social changes may occur in response to scenarios	Graph-Scrutinizer Graph-Greenifier Graph-Choreographer
Global news data analytics	• Develop tools for decision-makers to implement proactive policies for positive environmental impact and promote sustainable practices • Generate models for semantic understanding, sentiment analysis, and topic extraction • Distill large volumes of news into meaningful insights about current trends and even potential future events • Enable real-time monitoring of the Event Registry platform and evaluation of policy impacts	Graph-Scrutinizer Graph-Greenifier Event Registry platform

Functionality	Global news data ingestion
Requirement identifier	UC-2.1
Requirement name	Global news data ingestion
Priority	Very high
Description	Ingest, index and process JSON data from the Event Registry interface.
Precondition	Graph-Inceptor

Functionality	Global news data modelling
Requirement identifier	UC-2.2
Requirement name	Global news data modelling
Priority	Very high
Description	Model and test the ingested global news data using the Graph-Scrutinizer tool and help understand critical connections and gaps in constructing KGs.
Precondition	Graph-Scrutinizer; UC-2.1

Functionality	Global news data visualization
Requirement identifier	UC-2.3
Requirement name	Global news data visualization
Priority	Very high
Description	Visualise node and edge predictions on different UC scenarios.
Precondition	Graph-Scrutinizer; Graph-Greenifier; UC-2.2

Functionality	Global news data analytics
Requirement identifier	UC-2.4
Requirement name	Global news data analytics
Priority	Very high
Description	Process global news data sets ingested from the Event Registry interface to deliver valuable insights and predictions to the stakeholders and their UC scenarios.
Precondition	Graph-Scrutinizer; Graph-Greenifier; UC-2.1; UC-2.2; UC-2.3

A.3 UC-3: GREEN AI FOR A SUSTAINABLE AUTOMOTIVE INDUSTRY

Table 7 : UC-3 Green AI for a Sustainable Automotive Industry.

Functionality	Description	Tool
Manufacturing KG creation	Ingest sensor data and an ontology in the Graph-Massivizer platform and convert it to the KG format.	Graph-Inceptor
Quality assessment	Visualise statistics of the ingested graph data.	Graph-Scrutinizer
KG completion	Predict missing links in the KG.	Graph-Scrutinizer
Manufacturing KG preparation for AI	Embed (project) the KG in the latent space to further use AI models and store the generated vectors.	Graph-Scrutinizer
KG sampling	Generate a sample KG for proof-of-concept test cases.	Graph-Scrutinizer
Manufacturing KG processing	Measure the impact of AI algorithms on BOSCH CO_2 neutrality plans and provide energy scaling estimates.	Graph-Optimizer Graph-Greenifier Graph-Choreographer

Functionality	Manufacturing KG construction
Requirement identifier	`UC-3.1.1`
Requirement name	Manufacturing KG construction
Priority	Very high
Description	Ingest sensor data (CSV) and accompanying ontology in the Graph-Massivizer platform.
Precondition	Graph-Inceptor ; Sensor data availability; Manufacturing ontology

Functionality	Manufacturing KG creation and population
Requirement identifier	`UC-3.1.2`
Requirement name	Non-RDF data ingestion
Priority	Very high
Description	Ingest raw data (CSV) and convert it to the required KG format based on predefined rules.
Precondition	Graph-Inceptor ; `UC-3.1.1`

Functionality	Manufacturing KG quality assessment
Requirement identifier	`UC-3.2.1`
Requirement name	Manufacturing KG statistics
Priority	High
Description	Visualize the ingested manufacturing KG data statistics, such as relation counts, population, and missing information.
Precondition	Graph-Scrutinizer ; `UC-3.1.1`

Functionality	Manufacturing KG quality assessment
Requirement identifier	UC-3.2.2
Requirement name	Manufacturing KG data visualization
Priority	High
Description	Provide interfaces to visualize manufacturing sensors data and statistics, such as entities, sources, and related source files.
Precondition	Graph-Scrutinizer; UC-3.1.1

Functionality	Manufacturing KG completion
Requirement identifier	UC-3.3
Requirement name	Manufacturing KG link prediction
Priority	Medium
Description	Predict missing manufacturing KG links (e.g., linking to ontology).
Precondition	Graph-Scrutinizer; UC-3.1.1

Functionality	Manufacturing KG preparation for AI
Requirement identifier	UC-3.4
Requirement name	Embedding AI model training
Priority	Medium
Description	Automate the pipeline for training embedding AI models for the manufacturing KG and update it frequently.
Precondition	Graph-Scrutinizer; UC-3.1.1

Functionality	Manufacturing KG sampling
Requirement identifier	UC-3.5.1
Requirement name	Clustering algorithms for manufacturing KG sampling
Priority	Medium
Description	Provide graph clustering methods to discover communities in the manufacturing KG.
Precondition	Graph-Scrutinizer; UC-3.1.1

Functionality	Manufacturing KG sampling
Requirement identifier	UC-3.5.2
Requirement name	Manufacturing KG sample selection
Priority	High
Description	Generate a manufacturing representative KG sample based on the clustering results.
Precondition	Graph-Scrutinizer; UC-3.1.1; UC-3.5.1

Functionality	Manufacturing KG sampling
Requirement identifier	UC-3.5.3
Requirement name	Synthetic manufacturing KG sample generation
Priority	High
Description	Generate synthetic manufacturing KG samples with specific distributions that follow its ontology.
Precondition	Graph-Scrutinizer; UC-3.1.1; UC-3.5.1

A.4 UC-4: DATA CENTER DIGITAL TWIN FOR EXASCALE COMPUTING

Table 8 : UC-4 Data Center Digital Twin for Exascale Computing.

Functionality	Description	Tool
HPC monitoring data ingestion and mapping	Ingest real-time data and publicly available historical time series from the Examon data center monitoring tool in various formats such as Parquet files or databases.	Graph-Inceptor
HPC monitoring data management	• Enable data manipulation and management operations • Provide a user-friendly interface for displaying graphs for dealing with numerous nodes and edges • Save the graph in different formats for diverse needs	Graph-Inceptor Graph-Scrutinizer
HPC graph data analysis and integration with standard tools	• Perform clustering, classification and anomaly detection using GNNs, graph queries, and classical graph processing techniques integrated with popular libraries such as scikit-learn, scikit-network, PyTorch, TensorFlow, and Keras • Create data-driven graphs based on raw data or modify existing graphs (e.g. causality map)	Graph-Scrutinizer
Quantification of data center energy efficiency	• Enhance the sustainability of data centers through what-if analysis based on historical workload distributions • Provide tangible, explainable, and credible information about the energy efficiency and sustainability of complex workflows used by CINECA at local, national, and EU levels	Graph-Greenifier
UC workflow optimisation	Leverage holistic telemetry data and BGO properties to optimize the execution of complex UC workflows.	Graph-Optimizer
Computing continuum orchestration	Orchestrate BGO and Graph-Massivizer tools in the CINECA HPC computing continuum.	Graph-Choreographer

Functionality	HPC monitoring data ingestion
Requirement identifier	UC-4.1.1
Requirement name	Data center graph inception from real-time Examon monitoring data ingestion
Priority	Very high
Description	• Aggregate online Examon monitoring data as multivariate time series with correct granularity • Create an adjacency based on domain expertise • Monitoring topology, or ontology • Incept a KG suitable for ML models • Graph query and processing
Precondition	• HPC ontology • Integration of Graph-Inceptor with Examon online monitoring system

Functionality	HPC monitoring data ingestion
Requirement identifier	UC-4.1.2
Requirement name	Graph inception from Examon database integration
Priority	Very high
Description	• Automatically query relevant monitoring data from the Examon database using ExamonQL • Create a KG suitable for ML models • Graph query and processing
Precondition	• HPC ontology • Integration of Graph-Inceptor with historical Examon database

Functionality	HPC monitoring data ingestion
Requirement identifier	UC-4.1.3
Requirement name	Graph inception with public data repository integration
Priority	Very high
Description	• Aggregate monitoring data in different formats (e.g., CSV, Parquet) from public sources (Zenodo) • Incept a KG suitable for ML models • Graph query and processing
Precondition	• HPC ontology • Integration of Graph-Inceptor with public datasets (Zenodo)

Functionality	HPC monitoring data management
Requirement identifier	UC-4.2
Requirement name	HPC monitoring data visualization
Priority	Very high
Description	• Provide a user-friendly interface for displaying massive data • Saving them in various formats for different needs • Analyzing them (e.g., size and connectivity metrics)
Precondition	• HPC ontology • Integration of Graph-Inceptor with different datasets

Functionality	HPC graph data analysis and integration with standard tools
Requirement identifier	UC-4.3.1
Requirement name	Predictive ML models in a data center digital twin KG
Priority	High
Description	Create predictive ML models for the data center ICT and cooling infrastructure, including anomaly anticipation and room-level thermal evolution prediction.
Precondition	• Raw data and data center KG (UC-4.1.1, UC-4.1.2, UC-4.1.3) • GNN workload support in Graph-Scrutinizer

Functionality	HPC graph data analysis and integration with standard tools
Requirement identifier	UC-4.3.2
Requirement name	Descriptive data center digital twin models
Priority	Very high
Description	Create ML models for data-driven discovery of causal connections between metrics, features and higher-level events.
Precondition	• Raw data and data center KG (UC-4.1.1, UC-4.1.2, UC-4.1.3) • GNN and ML workload support in Graph-Scrutinizer

Functionality	Data center energy efficiency quantification
Requirement identifier	UC-4.4
Requirement name	Energy consumption optimization
Priority	Medium
Description	• Estimate energy costs and sustainability labels for different BGOs on the CINECA data center • Formulate energy savings and greener processing recommendations
Precondition	• Data center KG (UC-4.1.1, UC-4.1.2, UC-4.1.3) • Graph-Massivizer toolkit • Integration of Graph-Greenifier with Graph-Scrutinizer

A.5 ARCHITECTURE REQUIREMENTS

Deliverable D2.1 presents a detailed set of UC and Graph-Massivizer platform requirements resulting from the requirements elicitation phase. This section includes the platform requirements concerning the Graph-Inceptor and Graph-Scrutinizer tools.

Requirement identifier	GI-1
Requirement name	Graph generation
Priority	Very high
Description	Generate graphs in a format compatible with the downstream tools that use them.
Precondition	Graph-Inceptor

Requirement identifier	GI-2
Requirement name	Multiple data type support
Priority	High
Description	Support multiple input data types, including files, cloud data, and streams.
Precondition	Graph-Inceptor implementation

Requirement identifier	GS-1
Requirement name	Multiple graph views types
Priority	High
Description	Support different subgraph views to perform faster and more focused analytics on particular data of interest to a user.
Precondition	Graph-Inceptor, Graph-Scrutinizer

Requirement identifier	GS-2
Requirement name	Transparent reasoning methodology
Priority	High
Description	Implement a transparent methodology for reasoning and analysis to produce reliable results avoiding graph sampling biases.
Precondition	Graph-Inceptor, Graph-Scrutinizer

A.6 GRAPH-MASSIVIZER DATASET SUMMARY

Figure 22 summarises the input and generated datasets in the Graph-Massivizer platform, as listed in the deliverable D1.3.

Table 1: Summary of the input and generated data sets in the Graph-Massivizer platform.

Identifier	Title	Description	Type	Format	Size	Source	License	Personal data	Link
UC1.01	HEU commodities futures and USA stocks	Fundamental USA/EU commodities and technical data samples	Time series	CSV	~ [50 GB – 1TB]	Commercial	Proprietary	No	
UC2.01	Global foresight for environmental protection	News articles and events databases from over 300 thousand global publishers	Time series	JSON	~7 TB	Public	Proprietary	No	https://newsapi.ai/
UC3.01	Welding data	Large-scale welding quality data on car BiW assembly lines	Numeric	CSV	~1.2 GB per day	BOSCH factories	Proprietary	No	
UC3.02	Synthetic welding data	Smaller welding data samples for UC-3 simulation	Numeric	CSV	~1.5 GB	BOSCH factories	BOSCH factories	No	
UC4.01	M100 exadata	12 datasets of monitoring data (03.2020 – 09.2022)	Numeric time series	Parquet	454 GB	CINECA Examon	CC (Zenodo, open data)	No	https://gitlab.com/ecs-lab/exadata
UC4.02	M100 exadata KG	Monitoring data KG	Graph	TBD	TBD	UC4.01	Open data (Zenodo)	No	
GM.WIKI	Wikidata	Collaboratively edited multilingual KG	Graph	RDF	TBD	Public, collaboratively generated	CC (Open data)	No	https://www.wikidata.org/
GM.DIS	SemOpenAlex	Scientific publications KG	Graph	RDF	GB	OpenAlex	CC (Open data)	No	https://semopenalex.org/resource/

Figure 22: Grap-Massivizer dataset summary.

B ARCHITECTURE DESIGN

This section contains an excerpt of the architecture design from the deliverable D2.1 relevant to the work in WP3.

B.1 GRAPH-MASSIVIZER TOOLKIT ARCHITECTURE

The revised Graph-Massivizer toolkit architecture, shown in Figure 23 demonstrates how the tools and stakeholders interface and interact with each other into a combined integrated `Metaphactory` platform.

- The `Metaphactory` platform provides a graphical front-end for users to visualise and interact with the toolkit and to query the platform for information. Within the toolkit, complex operational pathways enable users to interact with separate tools or complex workflow combinations;
- The `Metaphactory` platform contains components that enable KG management, such as modelling, visualization, and graph editing.

A typical workflow enables complex operations that call each tool sequentially as a pipeline. This operational pattern enables various classic optimisations to improve performance, energy or other trade-offs. Common workflows and pipelines in a deployed toolkit configure each tool for running a default execution pipeline, with users potentially re-configuring (or modifying, for advanced users) parts depending on the UC.

- First in the pipeline would be the Graph-Inceptor, which the toolkit uses to initialize and handle massive graph ingestion and storage. Graph-Inceptor supports BGOs that create graphs and move them to the graph storage in coordination with the graph database when necessary;
- Graph-Scrutinizer uses this graph data, which supports BGOs to analyze and perform reasoning, including graph sampling and probabilistic reasoning;
- Graph-Optimizer optimizes the workflow by selecting the best-performing mix of BGO implementations for the workflow.
- Consequently, Graph-Greenifier analyses the graph data and processing metrics for performance and sustainability optimizations.
- Finally, the Graph-Choreographer leverages information provided by the performance and sustainability models to enable serverless BGOs and workflow processing by deploying on the compute infrastructure.

Each tool also has an internal log accessed by the other platform tools for general inspection, development and debugging. The final components external to the Graph-Massivizer toolkit, depicted to the left and bottom of the figure, include data sources and the computing infrastructure.

- Data sources are always accessible to the toolkit, independent of the pipeline and the graph created by Graph-Inceptor. Data providers can supply files, streams, databases, APIs for external graph resources, or anything that can store or represent graphs;

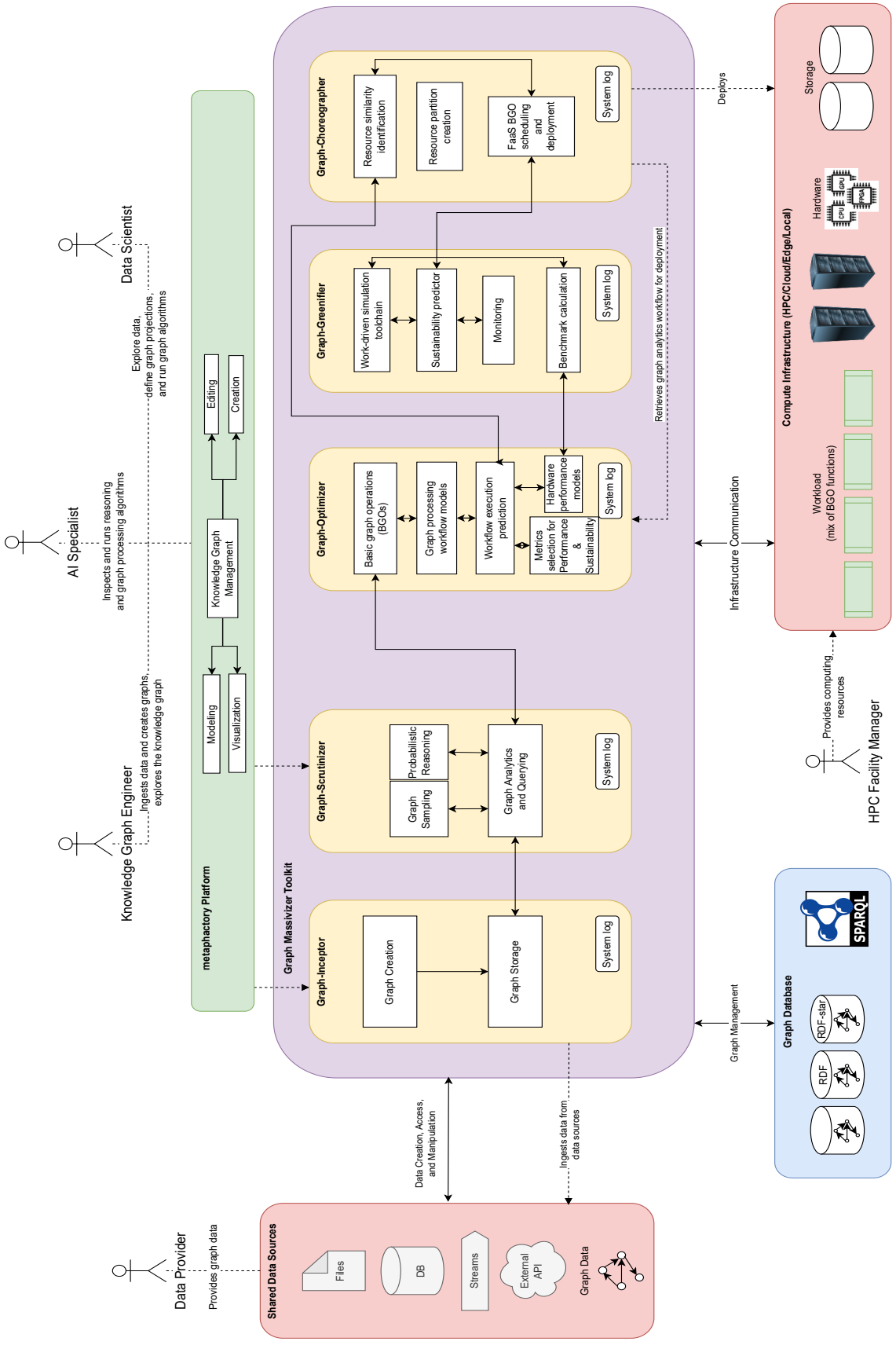


Figure 23: Graph-Massivizer architecture container diagram (level 2).

- All tools use the computing continuum ranging from high-performance computers, cloud and edge devices to traditional local machines with their own hardware, storage and graph processing workload allocation method;
- Graph database is a shared resource comprising graph storage with SPARQL query access and various formats (RDF and RDF-star) and representations used in the toolkit.

B.2 GRAPH-INCEPTOR ARCHITECTURE

The Graph-Inceptor tool [78], operated by a data engineer, facilitates the conversion of UC-specific data into extreme-scale graphs. These graphs encompass traditional types like directed, undirected, multi- and/or hypergraphs and their associated data structures, and virtual KGs. Virtual KGs represent a comprehensive composition of various graph types, with the capability of accessing a persistence layer to retrieve subgraphs or time-series data from databases.

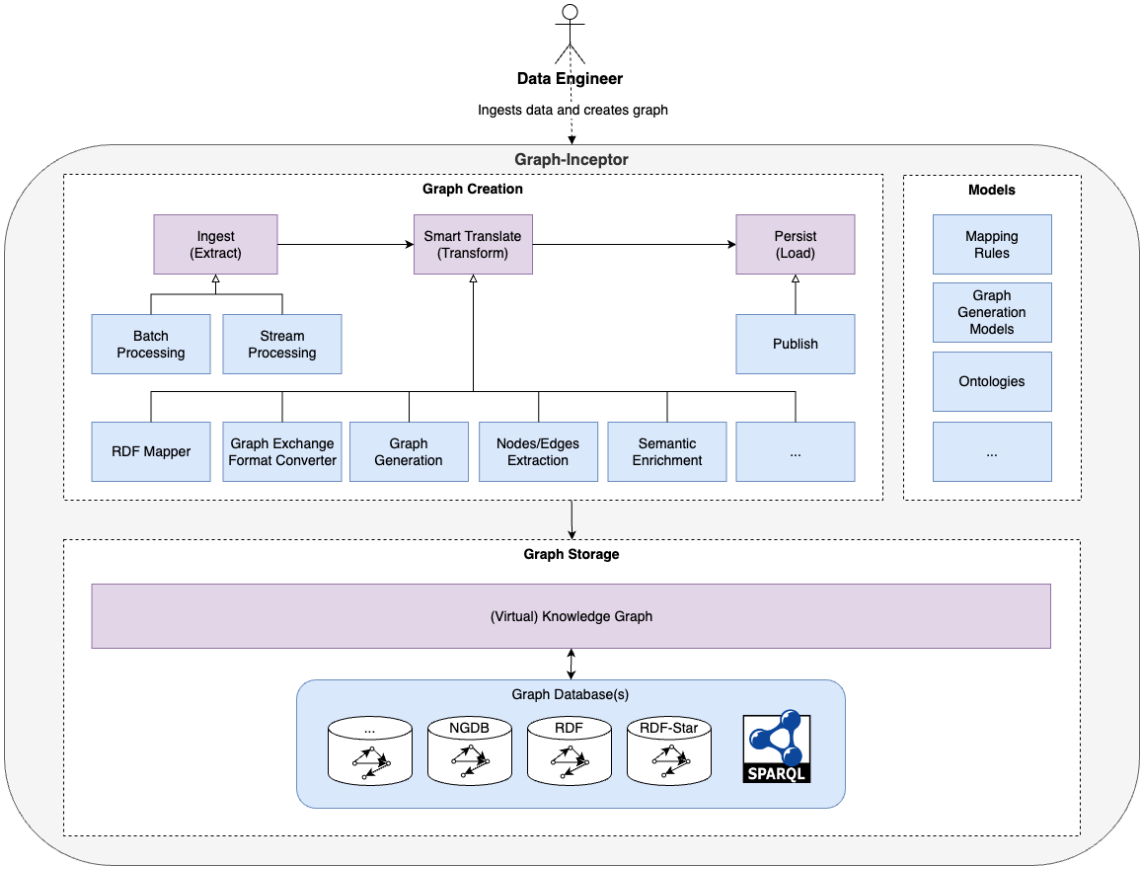


Figure 24: Graph-Inceptor architecture component diagram (level 3)

Graph-Inceptor consists of three components, shown in Figure 24:

- **Graph creation** implements BGOs to support the data ETL process;
- **Graph model** administrates graph generators, ontologies, and mapping rules for UC-specific datasets;
- **Graph storage** allows access to the storage layer through a virtual KG.

B.2.1 Graph Creation

The graph creation contains three components:

1. **Ingest** supports **batch processing** of data stored over a period of time, such as log files, and **stream processing** of data flows, such as event streams.
2. **Smart translate** provides features like RDF mapping and graph exchange formats conversions, such as GEXF [79], GraphML [80], or GML [81], graph data generation, nodes and edges extraction, and semantic enrichment. We envision a set of smart mapping rules, graph generation models, and ontologies that enhance Graph-Inceptor's ability to create realistic and diverse graph datasets for testing and validation purposes, as follows:
   - **AI translation** steps, using natural language processing and GNNs to extract relevant information from the data embedded into the graph or published as separate sub-graphs;
   - **ML-based generation** steps of appropriate synthetic graph data.
3. **Persist (Load)** publishes the graph data as a semantic (virtual) KG in the graph storage, referencing data scattered across different heterogeneous databases. For example, extremely large volumes of time series data from sensors or logs may not be suitable to be embedded into an RDF graph structure but instead stored in a different format (including ML and GNN models), referenced and queried for further analytics.

B.2.2 Graph Storage

Graph storage provides physical and virtual capabilities for the (virtual) KG using the **graph database** interface shown in Figure 23. In addition to RDF and RDF-Star triplestores, a neural graph database interface will enable efficient data storage, analytics and management.

ACRONYMS

Acronym	Definition
AI	Artificial Intelligence
AP	Average Precision
AUC	Area under the ROC Curve
AWS	Amazon Web Services
BC	Betweenness Centrality
BFS	Breadth-First Search
BGO	Basic Graph Operation
BSMA	Benchmark for Social Media Analytics
CDK	Cloud Development Kit
CLI	Command Line Interface
DFS	Depth-First Search
DG	Directed Graph
DIG	Dive into Graphs
DSE	Design Space Exploration
DTLB	First-level Data Translation Lookaside Buffer
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
ETL	Extract, Transform, and Load
GAP	GAP Berkeley
GBBS	Graph Based Benchmark Suite
GBTL	GraphBLAS Template Library
GCN	Graph Convolutional Neural Network
GDB	Graph Database Benchmark
GEXF	Graph Exchange XML Format
GML	Graph Modeling Language
GNN	Graph Neural Network
HPC	High Performance Computing
IAM	Identity and Access Management
IRI	Internationalized Resource Identifier
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
KG	Knowledge Graph
LDBC	Linked Data Benchmark Council Graphalytics
LPG	Labeled Property Graphs

Acronym	Definition
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multilayer Perceptron
MPKI	Misses per Kilo Instructions
MRR	Mean Reciprocal Rank
MST	Minimum Spanning Tree
NGDB	Neural Graph Database
NLP	Natural Language Processing
NN	Neural Network
OGB	Open Graph Benchmark
OWL	Web Ontology Language
PER	Peracton
R&D	Research and Development
RDD	Resilient Distributed Dataset
RDF	Resource Description Framework
RelBench	Relational Deep Learning Benchmark
REST	Representational State Transfer
RML	RDF Mapping Language
S3	Simple Storage Service
SaaS	Software-as-a-Service
SHACL	Shapes Constraint Language
SKOS	Simple Knowledge Organization System
SNAP	Stanford Network Analysis Project
SNS	Simple Notification Service
SSSP	Single Source Shortest Path
TCO	Total Cost of Ownership
TGB	Temporal Graph Benchmark
UC	Use Case
W3C	World Wide Web Consortium
WGB	Waterloo Graph Benchmark