D2.2: GRAPH-MASSIVIZER REQUIREMENTS ANALYSIS AND ARCHITECTURE DESIGN

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

Title	Document Version
Graph-Massivizer Requirements Analysis and Architecture Design	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*
M24 (December 2024)	24 January 2025	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	Organisation	Contributing WP
Radu Prodan	AAU	WP2

Authors	Organisation
Haleh Dizaji, Reza Farahani, Radu Prodan	AAU
Antonios Klironomos, Evgeny Kharmalov	BOSCH
Matteo Angelinelli, Giorgio Pedrazzi, Roberta Turra	CINECA
Gregor Leban	ER
Marko Grobelnik, Jože Rožanec	JSI
Aaron Eberhart, Peter Haase	MPH
Thomas Haselwanter, Laurenţiu Vasiliu	PER
Brian Elvesæter, Dumitru Roman, Daniel Schroeder	SINTEF
Andrea Bartolini, Junaid Ahmed Khan, Martin Molan	UNIBO
Ana Lucia Varbanescu	UTW
Ruud van Bakel, Michael Cochez, Alexandru Iosup, Tiziano de Matteis, Dante Niewenhuis, Sacheendra Talluri	VU

Abstract

Deliverable 2.2 presents the intermediate requirements analysis and the updated Graph-Massivizer toolkit and platform architecture design. It begins with describing the beta-testing campaign of the first toolkit prototype released at M12, comprising the methodology, commonalities and instantiation for each constituent tool. The campaigns' results with real beta testers gave fundamental input to the tools regarding their requirements evolution, architecture revision and subsequent development iterations. Recent state-of-the-art and market analysis revealed minimal changes to the use case requirements. The deliverable concludes with a revised architecture design of the operational Graph-Massivizer platform, focusing on the internal integration and communication among the tools and the external interaction with the metaphactory platform and its stakeholders.

This deliverable offers direct input to the upcoming follow-up deliverables D2.3, due at month 36, which will finalise the requirements analysis and validate the campaign results, including the final assessment of the key performance indicators.

Keywords

Architecture design; Basic graph operation; Beta-testing campaign; Graph database; Requirements analysis; BGO repository; Sequence diagram; BGO workflow.

REVISION HISTORY

Revision	Date	Description	Author (Organisation)
0.1	01.03.2024	Table of contents	Laurenţiu Vasiliu (PER)
0.2	01.04.2024	Beta-testing methodology, release status for all tools, and section assignment	Laurenţiu Vasiliu (PER)
0.3	01.05.2024	Beta-testing sections and content generation	Laurenţiu Vasiliu (PER)
0.4	03.06.2024	Beta-testing input by all tools	Ruud van Bakel (VU) Brian Elvesæter (SINTEF) Reza Farahani (AAU) Dante Niewenhuis (VU) Jože Rožanec (JSI) Daniel Schroeder (SINTEF) Ana Lucia Varbanescu (UTW)
0.5	16.09.2024	Internal draft version on all requirements sections	Laurenţiu Vasiliu (PER)
0.6	01.12.2024	First architecture revision	Jože Rožanec (JSI)
0.7	19.12.2024	Integration of requirements and beta-testing sections	Radu Prodan (AAU)
0.8	20.12.2024	Second architecture revision	Brian Elvesæter (SINTEF)
0.9	22.01.2024	Integration and consistency updates	Haleh Dizaji (AAU) Reza Farahani (AAU) Daniel Schroeder (SINTEF)
1.0	24.01.2024	Final quality assurance and submission	Haleh Dizaji (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 is available at https://graph-massivizer.eu/.

COPYRIGHT STATEMENT

INDEX

LIST OF FIGURES **7**

1 INTRODUCTION **10**

1.1 STRUCTURE OF THE DOCUMENT 11

2 GRAPH-MASSIVIZER BETA-TESTING CAMPAIGN **12**

2.1 BETA-TESTING METHODOLOGY 12
2.1.1 Beta-Testing Planning *12*
2.1.2 Beta-Testing Recruitment *12*
2.1.3 Beta-Testing Distribution *12*
2.1.4 Beta-Testing Execution *12*
2.1.5 Beta-Fixes and Improvements *13*
2.1.6 Beta-Testing Completion *13*
2.1.7 Expert Demonstration *13*
2.2 TOOL COMMONALITIES 13
2.2.1 General Questions *13*
2.2.2 Beta-Testing Evaluation *13*
2.2.3 Common Testing Metrics *13*
2.3 GRAPH-INCEPTOR 14
2.3.1 ETL Pipeline *14*
2.3.2 GraphMa *14*
2.4 GRAPH-SCRUTINIZER 15
2.4.1 ts2g2 ..... *15*
2.4.2 Graph-Scrutinizer *16*
2.5 GRAPH-OPTIMIZER 16
2.6 GRAPH-GREENIFIER 17
2.7 GRAPH-CHOREOGRAPHER 17

3 FIRST BETA-TESTING CAMPAIGN RESULTS **19**

3.1 GRAPH-INCEPTOR 19
3.1.1 ETL Pipeline *19*
3.1.2 GraphMa *24*
3.2 GRAPH-SCRUTINIZER 26
3.2.1 ts2g2 Library *26*
3.2.2 Graph-Scrutinizer *28*
3.3 GRAPH-OPTIMIZER 29
3.4 GRAPH-GREENIFIER 30
3.5 GRAPH-CHOREOGRAPHER 34
3.6 BETA-TESTING IMPLICATIONS 36
4.1 GRAPH-MASSIVIZER PLATFORM 37
4.1.1 Description and Mapping *37*
4.1.2 Graph-Massivizer Platform Documentation *37*
4.1.3 Graph-Inceptor Requirements Documentation *38*
4.1.4 Graph-Scrutinizer Requirements Documentation *39*
4.1.5 Graph-Optimizer Requirements Documentation *39*
4.1.6 Graph-Greenifier Requirements Documentation *40*
4.1.7 Graph-Choreographer Requirements Documentation *41*
4.2 USE CASES 41
4.2.1 Green and Sustainable Finance *42*
4.2.2 UC-2: Global Foresight for Environmental Protection *43*
4.2.3 UC-3: Green AI for Sustainable Automotive Industry *43*

4.2.4 UC-4: Data Centre Digital Twin for Sustainable Exascale Computing ... 44

5 GRAPH-MASSIVIZER ARCHITECTURE DESIGN ... 45

5.1 INTRODUCTION ... 45
5.2 TOOLKIT INTERACTION SEQUENCE DIAGRAMS ... 46
5.3 BGO REPOSITORY ... 47
5.3.1 BGO Model ... 47
5.3.2 BGO Implementation ... 47
5.3.3 BGO Ontology ... 48
5.3.4 BGO Registration ... 49
5.3.5 BGO Lookup ... 49
5.4 GRAPH DATABASE ... 50
5.4.1 Data Registration and Ingestion ... 50
5.4.2 Data Source Registration ... 50
5.4.3 ETL Pipeline Data Ingestion ... 51
5.4.4 GraphMa Data Ingestion ... 51
5.5 BGO WORKFLOW DEFINITION ... 52
5.6 BGO WORKFLOW EXECUTION ... 53
5.6.1 BGO Workflow Optimisation ... 54
5.6.2 BGO Workflow Greenifying ... 54
5.6.3 BGO Workflow Scheduling ... 56
5.6.4 BGO Workflow Deployment ... 56
5.6.5 BGO Workflow Execution ... 57

6 ACADEMIC KNOWLEDGE BGO WORKFLOW EXAMPLE ... 59

6.1 BGO WORKFLOW DEFINITION ... 59
6.2 BGO WORKFLOW EXECUTION ... 59

7 CONCLUSIONS ... 62

ACRONYMS ... 63

LIST OF FIGURES

Figure 1: Graph-Massivizer iterative development cycle. 10
Figure 2: Graph-Massivizer iterative requirements elicitation, analysis, definition, validation and evolution phases. 11
Figure 3: Graph-Inceptor ETL pipeline documentation rating. 20
Figure 4: Graph-Inceptor ETL pipeline relevant source data formats. 20
Figure 5: Graph-Inceptor ETL pipeline relevant source data types. 21
Figure 6: Graph-Inceptor ETL pipeline data partitioning schemes. 21
Figure 7: Graph-Inceptor ETL pipeline source data to RDF mappings relevant to own projects. 21
Figure 8: Graph-Inceptor ETL pipeline relevant hosting infrastructures. 22
Figure 9: Graph-Inceptor ETL pipeline relevant RDF databases. 22
Figure 10: Graph-Inceptor ETL pipeline incremental KG update support. 23
Figure 11: Graph-Inceptor ETL pipeline KG database update frequency. 23
Figure 12: Update visibility in the KG database. 23
Figure 13: Barriers hindering the ETL pipeline adoption. 23
Figure 14: Precompiled graph pipeline execution experience. 24
Figure 15: GraphMa graph pipeline composition and execution support. 25
Figure 16: Custom graph operator implementation experience. 25
Figure 17: Custom traverser operator implementation experience. 26
Figure 18: Graph-Scrutinizer ts2g2 library functionality evaluation (1: poor, 5: excellent). 27
Figure 19: Graph-Scrutinizer ts2g2 library times series length to graph size evaluation. 27
Figure 20: Graph-Scrutinizer ts2g2 library graph sizes to times series lengths evaluation. 27
Figure 21: Graph-Scrutinizer ts2g2 library generated time series evaluation (1: poor; 5: excellent). 28
Figure 22: Graph-Scrutinizer tested graph sizes. 28
Figure 23: Graph-Scrutinizer example execution difficulty (1: very difficult; 5: very easy). 29
Figure 24: Graph-Scrutinizer example clarity (1: unclear; 5: crystal clear). 29
Figure 25: Graph-Greenifier topology format and implementation understanding. 31
Figure 26: Graph-Greenifier medium topology file creation experience. 31
Figure 27: Graph-Greenifier scenario understanding. 32
Figure 28: Graph-Greenifier custom scenario creation experience. 32
Figure 29: Graph-Greenifier output file understanding. 32
Figure 30: Graph-Greenifier data aggregation understanding. 33
Figure 31: Graph-Greenifier average number of tasks determination experience. 33
Figure 32: Graph-Greenifier custom visualisation understanding from service output files. 33
Figure 33: Graph-Greenifier custom visualisation understanding from host output files. 33
Figure 34: Graph-Greenifier visualisation of pending tasks and carbon emissions experience. 34
Figure 35: Graph-Choreographer documentation quality and clarity experience. 35
Figure 36: Graph-Choreographer installation experience. 35
Figure 37: Graph-Choreographer installation template usefulness. 35
Figure 38: Graph-Choreographer sample workflow relevance and experience. 35
Figure 39: Graph-Choreographer workflow scheduling performance satisfaction. 35
Figure 40: Graph-Choreographer execution mode preferences. 36
Figure 41: Graph-Choreographer execution module user experience. 36
Figure 42: Graph-Choreographer recommendation intent (1: do not recommend, 5: highly recommend). 36
Figure 43: Conceptual high-level overview of the Graph-Massivizer architecture. 45

Figure 44: C4 context diagram of the Graph-Massivizer toolkit architecture. 46
Figure 45: Graph-Massivizer toolkit external interaction sequence diagram. 47
Figure 46: BGO repository and the five Graph-Massivizer tools. 47
Figure 47: BGO ontology diagram. 48
Figure 48: BGO registration sequence diagram. 49
Figure 49: BGO lookup sequence diagram. 49
Figure 50: Data registration and ingestion sequence diagram. 50
Figure 51: Data source registration sequence diagram. 50
Figure 52: ETL pipeline data ingestion sequence diagram. 51
Figure 53: GraphMa data ingestion sequence diagram. 52
Figure 54: BGO workflow definition sequence diagram. 52
Figure 55: BGO workflow execution sequence diagram. 53
Figure 56: BGO workflow optimisation sequence diagram. 54
Figure 57: BGO workflow greenifying sequence diagram. 55
Figure 58: BGO workflow scheduling sequence diagram. 56
Figure 59: BGO workflow deployment sequence diagram. 57
Figure 60: BGO workflow execution sequence diagram. 58
Figure 61: Academic knowledge BGO workflow. 59
Figure 62: Academic knowledge BGO workflow sequence diagram. 60

LIST OF TABLES

Table 1: Tools beta-testing feedback summary. 19
Table 2: Graph-Inceptor ETL pipeline beta-testing feedback summary....20
Table 3: Graph-Inceptor beta-testing GraphMa feedback summary....24
Table 4: Predefined graph pipeline experience. 24
Table 5: Graph pipeline composition and execution experience....25
Table 6: Custom graph operator implementation feedback....25
Table 7: Custom traverser operator implementation experience....26
Table 8: Custom data sources and graph processing pipeline experience. 26
Table 9: Graph-Scrutinizer ts2g2 library beta-testing feedback summary. 26
Table 10: Graph-Scrutinizer beta-testing feedback summary....28
Table 11: Graph-Optimizer BGO workflow specification feedback. 30
Table 12: Graph-Optimizer hardware configuration and micro-benchmarking feedback. .30
Table 13: Graph-Optimiser prediction feedback. 30
Table 14: Graph-Optimiser graph properties feedback....30
Table 15: Graph-Greenifier simulation setup beta-testing feedback summary. 31
Table 16: Graph-Greenifier beta-testing feedback summary....31
Table 17: Graph-Choreographer workflow scheduling beta-testing feedback summary. ...34
Table 18: Graph-Choreographer execution engine beta-testing feedback summary....34
Table 19: Implications of beta-testing campaigns tool development....36
Table 20: UC-1 details: green and sustainable finance. 42
Table 21: UC-2 details: global foresight for environmental protection. 43
Table 22: UC-3 functionality mapping to Graph-Massivizer tools. 43
Table 23: Graph-Massivizer toolkit external interaction steps. 47
Table 24: BGO registration steps....49
Table 25: BGO implementation lookup steps....49
Table 26: Data registration and ingestion steps. 50
Table 27: Data source registration steps. 51
Table 28: ETL pipeline data ingestion steps....51
Table 29: GraphMa data ingestion steps....52
Table 30: BGO workflow definition steps....53
Table 31: BGO workflow execution steps....53
Table 32: BGO workflow optimisation steps....54
Table 33: BGO workflow greenifying steps....55
Table 34: BGO workflow scheduling steps....56
Table 35: BGO workflow deployment steps. 57
Table 36: BGO workflow execution steps....58
Table 37: Academic knowledge BGO workflow execution steps. 61

1 INTRODUCTION

Graph-Massivizer defines its **requirements** as a set of specific functionalities of its platform and tools needed by stakeholders that guide the design and implementation phases. The procedure for defining requirements has a four-step iterative methodology, described in **deliverable D2.1**, which includes requirements elicitation, analysis and specification, validation and evolution. Figure 1 shows these methodological steps integrated into the Graph-Massivizer iterative development across its WPs.

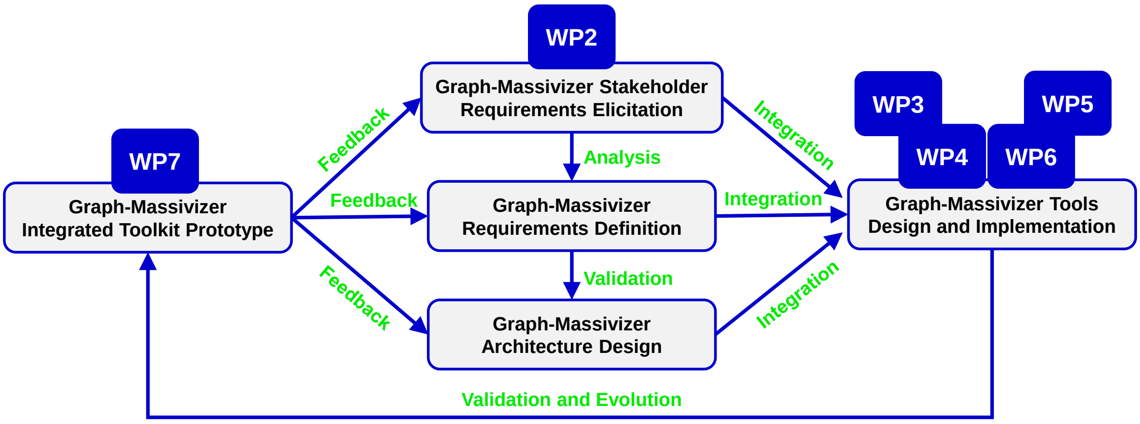

Figure 1: Graph-Massivizer iterative development cycle.

The evolution of requirements is the process of managing changes that arise during the collection of new requirements as the project develops. In this continuous phase, the consortium periodically iterates through the first three steps of the requirements definition methodology, following an agile state-of-the-art approach, to monitor and accommodate any potential changes. The evaluation process involves assessing each identified or proposed change compared to the existing requirements and the architecture of the Graph-Massivizer platform. Furthermore, it carefully analyses the trade-off between the cost and benefit of implementing a change before deciding to approve or reject it.

Graph-Massivizer consists of **four requirements elicitation and evolution phases**, concluding with a significant **milestone**, as follows:

- **MS-1: Requirement elicitation** concluded a state-of-the-art analysis at M3, extracted user stories from interviews with relevant internal and external stakeholders, and conducted brainstorming sessions with the focus group experts, as documented in the deliverable D2.1.
- **MS-2: Preliminary Graph-Massivizer tool prototypes** successfully concluded at M15 in two sequential steps:
    1. **First development iteration** concluded at M12 led to the toolkit architecture design based on detailed requirements analysis and released a preliminary version of the tools in an internal GitHub repository[1].
    2. **First validation campaigns** with external beta-testers and focus group members collected feedback using individual tools, summarised it as part of the technical periodic report delivered at M18 and documented it as part of this deliverable (see Section 2).
- **MS-3: First validated Graph-Massivizer toolkit prototype** targeted at M27 results again from two steps.

[1] https://github.com/graph-massivizer

1. **Second development iteration** incorporated the beta-tester and other stakeholder feedback (see Section 3), evolved the requirements (see Section 4), revised the toolkit architecture at M24 and as part of this deliverable (see Section 5), and will release a first alpha-tested integrated toolkit prototype at M27.
2. **Second validation campaigns** conducted with external beta-testers and focus group members will start at M33, collect feedback using the integrated toolkit and prototype UCs, and release the final validated and integrated toolbox at M36.

- **MS-4: Final validated Graph-Massivizer toolkit** targeted at M36 will consist of three steps:
  1. **Third development iteration** will incorporate the beta-tester and other stakeholder feedback, evolve the requirement again, revise the toolkit architecture and release a second beta-tested integrated toolkit prototype at M30.
  2. **Third validation campaigns** conducted with external beta-testers and focus group members will evaluate the KPIs associated with the project's technical objectives and operational UCs at M30.
  3. **Release of the Graph-Massivizer toolkit** as beta-tested open-source software will represent the end of the project and the start of the follow-up joint ventures targeting its commercialisation at M36.

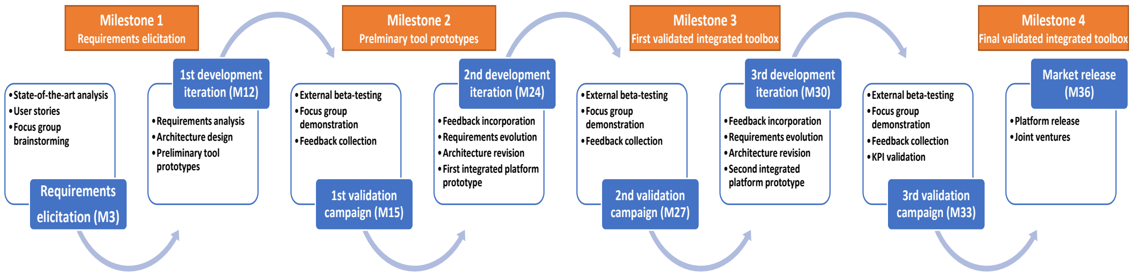

Figure 2: Graph-Massivizer iterative requirements elicitation, analysis, definition, validation and evolution phases.

1.1 STRUCTURE OF THE DOCUMENT

The deliverable has seven sections:

- Section 1 introduces the deliverable and the current status in the iterative prototype beta-testing, requirements evolution and architecture design;
- Section 2 presents the methodology of the beta-testing validation campaigns followed by the Graph-Massivizer tools;
- Section 3 summarises the main feedback and results received from the beta-testing validation campaign for each tool;
- Section 4 evolves the requirements of the Graph-Massivizer tools and platform according to the received beta-testing feedback;
- Section 5 evolves the Graph-Massivizer platform into an integrated architecture defining the tool interfaces and workflow tools according to the updated requirements;
- Section 6 exemplifies the interaction of the Graph-Massiviezr tools in the context of an open-source academic knowledge pilot use case;
- Section 7 concludes the deliverable and presents future developments.

2 GRAPH-MASSIVIZER BETA-TESTING CAMPAIGN

The consortium designed a high-level methodological template and plans to follow it for beta-testing the initial tool prototypes presented in Section 2.1. The beta-testing interaction has common aspects presented in Section 2.2 and tool-specific ones, detailed afterwards.

2.1 BETA-TESTING METHODOLOGY

The overall beta-testing methodology of the Graph-Massivizer tools follows six steps.

2.1.1 Beta-Testing Planning

- **Goal and objective** definition in the specific areas that require testing and desired outcomes involving individual tools beta-testing and Graph-Massivizer platform for all use cases;
- **Beta-testing type** definition, including:
  - **Closed beta-testing**, limited to a selected and controlled user group, typically internal teams or trusted partners;
  - **Focused beta-testing**, concentrated on specific features and functionalities, aiming for in-depth testing;
  - **Community beta-testing** engaging external active users for ongoing feedback throughout the development lifecycle;
- **Beta-testing selection** criteria involving target, use patterns, and technical expertise;
- **Beta-testing planning** involving a schedule, communication channels, and feedback mechanisms;

2.1.2 Beta-Testing Recruitment

- **Beta-tester pool selection** defining criteria and use patterns combining trust, technical expertise, and work relationship;
- **Recruitment strategy** involving partners, clients, or social media followers driven by each tool;
- **Beta-tester agreement** involving clear testing guidelines, confidentiality requirements, and feedback expectations;

2.1.3 Beta-Testing Distribution

- **Beta-software distribution**, providing access to testers with proper installation and use instructions;
- **Feedback collection**, implementing tools such as bug reporting and survey forms following generic questions and tool-specific templates to capture input;
- Communication channels for testers to ask questions, submit feedback, and receive updates, such as internal networks, emails, shared documents, and Google forms;

2.1.4 Beta-Testing Execution

- **Tester instructions**, providing comprehensive guidance on using the software and tasks to perform;
- **Testing activities**, guiding testers through various usability, functionality, performance, and compatibility scenarios;
- **Feedback analysis**, regularly reviewing the collected feedback to identify recurring issues, patterns, and user suggestions;
- **Intellectual property and data security** protection of sensitive information;
- **Legal and ethical considerations** concerning data privacy, intellectual property, and user agreements;
- **Analysis of key metrics per tool** to avoid duplication with the project KPIs, such as bugs, feedback related to functionality and performance, as well as user experience and engagement to assess testing effectiveness;

2.1.5 Beta-Fixes and Improvements

- **Feedback prioritisation** based on severity, frequency, and impact of user experience;
- **Bug fixes** ensuring compatibility with other components and tools;
- **Usability enhancements** addressing issues like confusing interfaces, inefficient workflows, and unclear instructions;
- **Performance optimisation** by improving code, memory use, data storage, and others.

2.1.6 Beta-Testing Completion

- **Final feedback collection** addressing remaining concerns and suggestions;
- **Tool release**, making final adjustments and finalising the software;
- **Tool documentation**, keeping a log of changes and fixes;

2.1.7 Expert Demonstration

- **Advisory board member** engagement and feedback collection.

2.2 TOOL COMMONALITIES

The first beta-testing stage addresses the tools in isolation, does not cover their integration and interaction, and assumes that their inputs and outputs are compatible.

2.2.1 General Questions

The consortium designed a set of high-level questions applicable to any tools:

- *How easy are the installation instructions to follow?*
- *How easy is running the basic example indicated in the tutorial (e.g., BFS of UC-0)? Are the results (clearly) displayed and easy to interpret?*
- *How easy is it to apply the library to the use case example?*
- *How do you rate the documentation? Did you encounter inaccuracies in the documentation of the basic example? What aspects of the documentation need improvement?*
- *What functionality is missing for the next release?*
- *What did you like most and least about the tool?*
- *What enhancements to the interface or benchmark could improve their intuitive use?*

2.2.2 Beta-Testing Evaluation

Each scenario asks the beta-tester to install and perform several tasks using the tool or component under investigation using a template example and following detailed instructions in a README file in the GitHub repository. After completing each scenario, the beta-tester must answer several questions about functionality, performance, user experience and bugs. The questionnaire contains a set of sections that describe a task or scenario for the beta tester. Each task and scenario define a set of questions asking to rank from *"strongly disagree"* to *"strongly agree"*, concluded with additional free text feedback.

2.2.3 Common Testing Metrics

- Functionality-related
  - Functional coverage in percentage, addressing the requirements.
- User experience-related
  - GitHub engagement, quantified in the number of stars, active users and contributors;
  - Documentation of searching, finding answers and understanding answers;
  - Usability[2] measured in ISO metrics concerning learnability, efficiency, memorability, errors and satisfaction;
  - Suggestions for improvements;
- Bugs-related
  - Number of test cases;
  - Number of successful and failed (unit) test cases;

[2] https://en.wikipedia.org/wiki/Usability

- Number of bugs reported, accepted, corrected, rejected and deferred, categorised into severe, critical annoyance and wish list.

2.3 GRAPH-INCEPTOR

Graph-Inceptor[3] uses Google Forms[4] to create a questionnaire for the ETL pipeline and GraphMa components. The first beta testing campaign focused on functionality rather than performance.

2.3.1 ETL Pipeline

Beta-testers

- 15 beta-testers: Enterprise IT consultants and potential users of an ETL pipeline for their customer or development projects.

Knowledge prerequisites

- Access to the ETL pipeline GitHub repository[5];
- AWS account, command line interface and Cloud development kit (TypeScript);
- Docker for running and building local images;
- RDF semantics for RML mappings;
- Structured data input (CSV, JSON, XML);
- Ontotext GraphDB (RDF database).

Testing scenarios

1. Setup ETL pipeline;
2. Use the public "Dimensions Covid" dataset example, prepared and ready to execute for evaluation and testing;
3. Use a custom dataset and follow the documentation on data preparation, RDF mappings definition, and ingestion.

Metrics

- Dataset size
  - Number, size, and format of Input files;
  - RDF statement count;
- Pipeline runtime for a given dataset
  - Overall runtime;
  - Runtime of individual steps, especially (the longest-running) RDF conversion, database ingestion, and full-text indexing;
  - Maximum RDF conversion concurrency (increasing with the number of input files).

2.3.2 GraphMa

Beta-testers

- 6 – 7 beta-testers: one from each use case partner (PER, ER, BOSCH, CINECA), at least two from SINTEF and one from MPH.

Knowledge prerequisites

- Access to the GraphMa GitHub repository[6];
- Java Development Kit or IDE (e.g., IntelliJ IDEA Community Edition)[7];
- Java experience (i.e., write, compile and run), particularly for scenarios 3 and 4.

---

[3] https://github.com/graph-massivizer/graph-inceptor
[4] https://docs.google.com/forms/d/1jRUmwed8Yk3uzINot7F-uPK8dDa8pRpxhKsFx_eG_ik/prefill
[5] https://github.com/graph-massivizer/graph-inceptor-etl-pipeline
[6] https://github.com/graph-massivizer/graph-inceptor-graphma
[7] https://www.jetbrains.com/idea/download/

Testing scenarios

1. Run several predefined graph pipelines through a command line interface (CLI) using a precompiled GraphMa library. The README file comprises:
   - A CLI command for each graph pipeline;
   - Input data set for each graph pipeline of different sizes to provide performance feedback.
2. Compose and run a custom graph processing pipeline from existing operators provided by the GraphMa library. The README file comprises:
   - Natural language description and instructions of the pipeline to write;
   - Instructions on writing and compiling a Java function that composes the graph operations into a GraphMa pipeline;
   - Code snippet to fill in and use by the beta-tester;
   - Instructions on finding the correct composition operators based on the GraphMa documentation;
   - Instructions on running the pipeline with the provided sample datasets.
3. Implement a custom traversal operator for a specific input data structure (e.g. CSV file). The README file comprises:
   - Instructions on traverser writing starting from a code snippet and compilation rules;
   - Instructions on testing the traverser using test graphs from the GraphMa repository.
4. Implement a custom graph algorithm operator wrapping a data structure and graph algorithm using the JGraphT[8] Java library. The README file comprises:
   - Instructions on operator writing starting from code snippet and compilation rules;
   - Instructions on testing the operator using test graphs from the GraphMa repository.
5. Use the learnings for specific use case-related data sources and graph processing pipelines (optional).

Testing metrics

- Number of supported BGOs and categories (e.g., traversal, algorithm, file);
- Scalability with input data sets involving different file formats and sizes, number of nodes and edges and graph density;
- BGO processing time and speed in the number of nodes or edges per second.

2.4 GRAPH-SCRUTINIZER

Graph-Scruzinizer[9] uses Google Forms to create a questionnaire to assess the two libraries and the first version of a graph sampling benchmark. This first beta-testing campaign focused on testing functionality and user experience.

2.4.1 ts2g2

Beta-testers

- 4 – 7 beta-testers: 3 – 5 from the use case partner (PER, ER, BOSCH, CINECA) and one or two from JSI and VU.

Knowledge prerequisites

- Access to the ts2g2 GitHub repository[10];
- Basic Python programming skills.

Testing scenarios

1. Create graphs from time series using a predefined Jupyter notebook, as follows:
   - Provide time series relevant to the use cases;
   - Persist the resulting graphs and perform basic analysis;
2. Create time series from graphs using a predefined Jupyter notebook, as follows:

[8] https://jgrapht.org/
[9] https://github.com/graph-massivizer/graph-scrutinizer
[10] https://github.com/graph-massivizer/ts2g2

    - Provide the graphs created in the scenario 1;
    - Persist the resulting time series and perform basic analysis;
3. Use the learnings to implement a custom use case analysis or functionality (optional).

Testing metrics

- Scalability for increasing graph size;
- Processing speed in node, edges, or data items per second;
- Graph and time series quality.

2.4.2 Graph-Scrutinizer

Beta-testers

- 3 – 6 beta-testers: 1 – 3 from the use case partner (PER, ER, BOSCH, CINECA) and two from JSI and VU.

Knowledge prerequisites

- Access to the Graph-Scrutinizer GitHub repository[9];
- Basic Python programming skills.

Testing scenarios

1. Install the Graph-Scrutinizer tool using the instructions in the README.md file and report on its success by filling in the scenario-specific questionnaire;

2. Run the Graph-Scrutinizer tool following the steps specified in the README.md file and report by filling in the scenario-specific questionnaire.

Testing metrics

- Scalability for increasing graph size;
- Processing speed in node, edges, or data items per second;
- Graph and time series quality.

2.5 GRAPH-OPTIMIZER

Graph-Optimizer[11] separately beta-tests four of its components, introduced in deliverable D6.1, covering GO-Data, comprising data extraction and preprocessing, GO-BGOs, comprising BGO modelling and benchmarking, GO-uBench, comprising micro benchmarking suite execution, and GO-predict comprising the prediction model combining the three components. The beta-testing assesses whether the models' construction, calibration, and use are precise.

Beta-testers

- 5 staff colleagues and PhD students at UTW.

Knowledge prerequisites

- Access to the Graph-Optimizer GitHub repository[11];
- Remote access to an HPC system for native execution;
- Basic knowledge of running containerised software;
- Basic understanding of performance and energy consumption metrics.

Testing scenarios

1. Assess the workflow specification, reading and creation and suggest improvements by:
    - Testing the workflow;
    - Modifying the workflow to implement additional functionality;
2. Assess hardware characterisation using micro-benchmarking and suggest improvements by:
    - Reading the current hardware configuration and run micro-benchmarking;

[11] https://github.com/graph-massivizer/graph-optimizer

  - Altering the hardware configuration and rerun micro-benchmarking;
3. Assess the functionality of the performance and energy predictive models and suggest micro-benchmarking improvements by:
   - Executing the original workflow and system and interpreting the result;
   - Assessing a specific BGO performance;
   - Changing the micro-benchmark and re-assess the BGO performance;
4. Observe sensitivity to changing graph properties by altering graph characteristics (edges and vertices) and observe the impact.

Testing metrics

- Micro-benchmarking functionality;
- Prediction functionality (i.e., output value), insight per BGO and
- Prediction sensitivity to changes in micro-benchmarking;
- Difficulty of changing the graph predictably;
- BGO prediction results for different graph topologies.

2.6 GRAPH-GREENIFIER

Graph-Greenifier[12] uses Google Forms to create a questionnaire that evaluates the scheduling component of the tool by conducting multiple input trials to verify the accuracy of input processing, calculation precision, and overall performance.

Beta-testers

- 10 beta-testers: 5 PhD and MSc students at UTW and VU and 5 consortium members.

Knowledge prerequisites

- Access to the Graph-Greenifier GitHub repository[12];
- Basic Python programming skills;
- Basic understanding of simulation;
- Basic understanding of data analysis and visualisation.

Testing scenarios

A README file provides beta-testing instructions to:

1. Execute a basic simulation, learn the required input and format to run a simulation and provide feedback on clarity and ease of use;
2. Create a new simulation using different input files and provide feedback on the ease of creating and changing them;
3. Simulate workloads in different data centres, analyse and visualise the large amounts of generated data to get proper insights and provide feedback on using the analysis and visualisation functionalities, highlighting missing ones.

Testing metrics

- Function execution time;
- Workflow completion time;
- Overall economic cost;
- Energy consumption.

2.7 GRAPH-CHOREOGRAPHER

Graph-Choreographer[13] uses Google Forms to create a questionnaire that evaluates the scheduling component of the tool by conducting multiple input trials to verify the accuracy of input processing, calculation precision, and overall performance.

Beta-testers

- 20 beta-testers: 15 AAU students and 5 staff colleagues.

[12] https://github.com/graph-massivizer/graph-greenifier
[13] https://github.com/graph-massivizer/graph-choreographer

Knowledge prerequisites

- Access to the Graph-Choreographer GitHub repository[13];
- Basic Python programming skills;
- Basic Cloud computing skills.

Testing scenarios

A README file provides beta-testing instructions to:

1. Execute a basic tool installation, including the required libraries;
2. Execute sample BGO workflows using predefined inputs and show the resulting execution plans and measured metrics;
3. Run a custom workflow from an offered selection with diverse input options in single or batch mode with varying testbed workloads and show the resulting execution plans and measured metrics.

Testing metrics

- Function execution time;
- Workflow completion time;
- Overall economic cost;
- Energy consumption.

3 FIRST BETA-TESTING CAMPAIGN RESULTS

Table 1 summarises the feedback received from the first beta-testing campaign, detailed in the following subsections.

Table 1: Tools beta-testing feedback summary.

Tool	Released functionality	Pending functionality
Graph-Inceptor	• ETL pipeline features complete and tested; • GraphMa basic functionality and scaffolding framework (to extend according to use case environments).	• GraphMa `LoadNetwork` BGO missing standard database connectors; • Filter implementations to customisable filter operators; • More `Data2Graph` BGO input formats (XML, CSV, JSON, Parquet); • Examon database integration in the `Interface2Graph` BGO; • More standard database connectors in the `Store2Graph` BGO; • Support for established graph formats in the `Graph2File` BGO.
Graph-Scrutinizer	• Go-Network library with graph sampling and summarisation algorithms; • Ts2g2 library with time series to graphs and graphs to time series transformation algorithms.	• Increased support and scalability; • Support for Cloud, Edge and HPC environments.
Graph-Optimizer	• In-progress components: GO-Data, GO-BGO, GO-uBench, GO-predict and GO-generate; • Script-based integrated tool release.	• Automated calibration; • Automated benchmarking and validation; • Component integration in a coherent tool; • Interface connection to Graph-Choreographer and Graph-Greenifier.
Graph-Greenifier	• Stand-alone tool release with improved execution and configuration interaction; • In-progress interface to Graph-Optimizer.	• Support for more complex workflows; • Interface connection to Graph-Optimizer and Graph-Choreographer.
Graph-Choreographer	• Time and cost-aware orchestration logic and optimisation algorithm; • Computing continuum support.	• Energy-aware optimisation algorithms; • Monitoring system; • Complex workflow integration and testing • Larger testbed support with increasing nodes and functionalities.

3.1 GRAPH-INCEPTOR

Graph-Inceptor conducted independent beta-testing campaigns for the ETL pipeline and GraphMa library.

3.1.1 ETL Pipeline

The beta testing survey for the ETL pipeline yielded 15 responses from mainly enterprise IT consultants and, hence, potential users describing usage scenarios for their respective customers or development projects. The use scenarios are mostly about enterprise data integration, bridging silos by integrating data from various sources and creating KGs. The ETL pipeline supports this use case with a framework describing the concepts and options space and a concrete implementation for a specific scenario and environment. Table 2 summarises the received beta-testing feedback.

Table 2: Graph-Inceptor ETL pipeline beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Inceptor ETL pipeline.
Participants	15 enterprise IT consultants.
Topics	Use scenario, documentation, data sources, semantic data model, tooling support, deployment, support, and feedback.
Feedback summary	• Complete functionality; • Missing support for other types of deployment; • Open question on RML mappings capturing all relevant aspects; • Complex RDF mappings requiring some effort; • KG design, requiring hours or days to create; • Compliance needs consideration.

Pipeline documentation

The documentation rating, comprising the concepts, architecture and setup, was generally good or at least sufficient, although the setup and navigation within the documentation could use some improvements. The beta testers requested more information on technical and implementation details and the steps to customise the ETL pipeline.

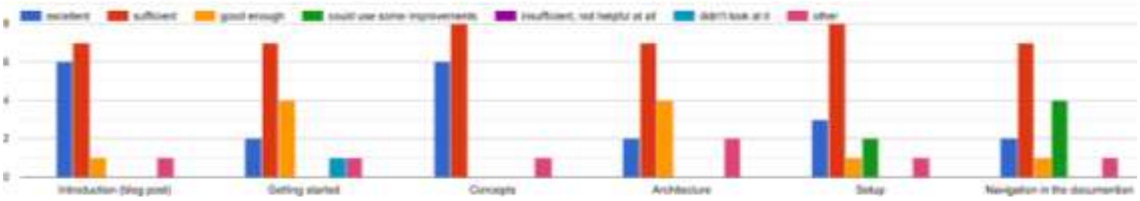

Figure 3: Graph-Inceptor ETL pipeline documentation rating.

Source data

The source data formats (Figure 4), types (Figure 5), and volumes highly vary over the described scenarios and cover the most relevant requirements. The data volume ranges from a few megabytes to gigabytes or even petabytes and from a few thousand records or entities to hundreds of millions, as required by most enterprise scenarios. The ETL pipeline implementation run on AWS created, in one instance, a KG with 33 billion RDF statements and 3 TB of data. The pipeline partitions the source data into multiple files by various dimensions, such as entity type, time, size, or data source (Figure 6).

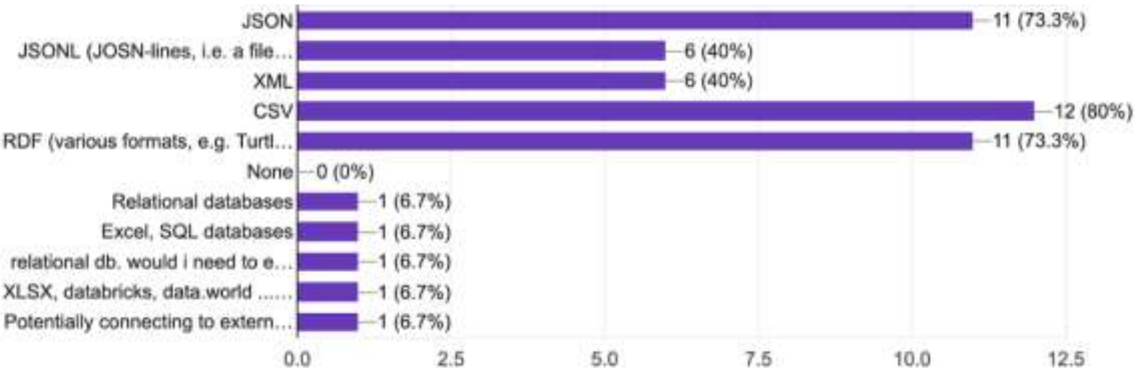

Figure 4: Graph-Inceptor ETL pipeline relevant source data formats.

However, the participants asked for support of Excel worksheets, relational databases, and data streams besides CSV files. While the ETL pipeline does not directly support relational databases to avoid data partitioning, it can export and ingest data from database queries into a supported format. Data retrieved via REST calls falls under the same restriction.

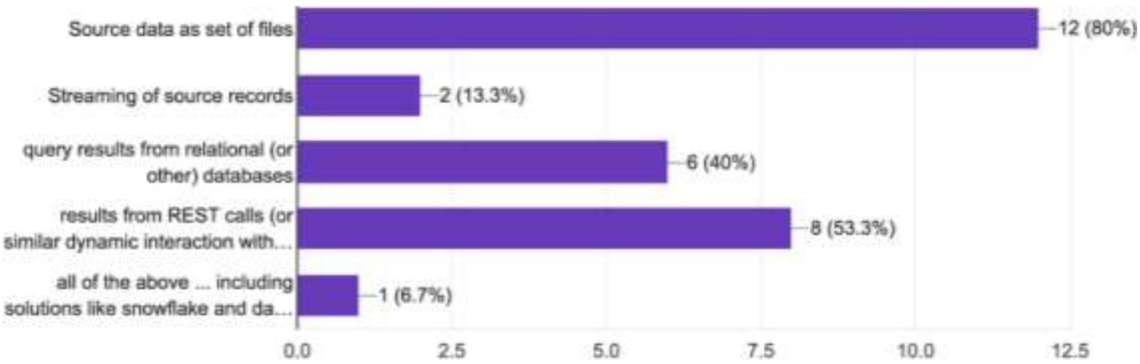

Figure 5: Graph-Inceptor ETL pipeline relevant source data types.

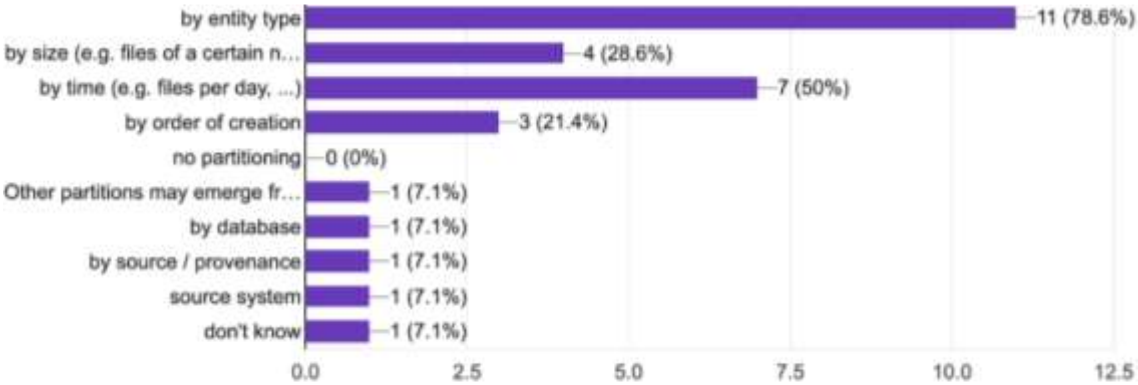

Figure 6: Graph-Inceptor ETL pipeline data partitioning schemes.

Semantic data model

The semantic data models used in the described scenarios are of average complexity, with a small number of ontologies and a medium number of classes, properties and concepts. Multiple ontologies maintained by different groups combine data from independent sources to fit the integrated model. The ETL pipeline provides comprehensive support for declarative mappings of source data to RDF using RML and R2RML. However, whether declarative mappings RML are insufficient to capture all relevant aspects of the conversions and incremental updates is unclear. The effort for data modelling and creating RDF mappings ranges from a few days to significant time (a few weeks), mitigated by following the iterative process outlined in the conceptual documentation. The support for programmatic mapping complementing the declarative ones needs improved documentation on customising the pipeline. More mapping examples and documentation will lower the learning curve.

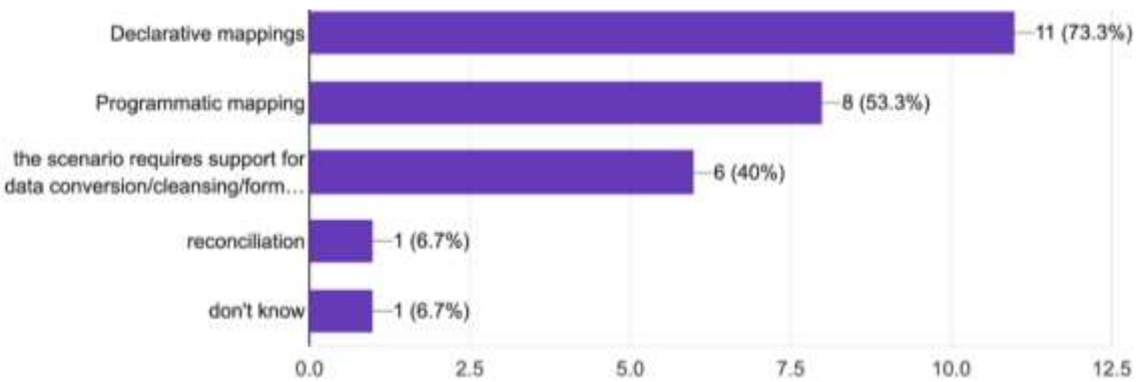

Figure 7: Graph-Inceptor ETL pipeline source data to RDF mappings relevant to own projects.

Tooling support

metaphactory as an ontology modelling tool comprehensively covers the authoring of semantic data models. However, the responders had tooling requests external to the ETL pipeline for editing RML mappings, as authoring them is complex and error-prone. At a minimum, visualisation and cross-referencing RML mappings with the ontologies show that dependencies are necessary. Additionally, integration with visualisation of data validation results will indicate the mappings to check in case of data issues and violations. Finally, tooling for monitoring could provide frequent missing insights and performance metrics regarding pipeline operations' status, progress and issues.

Pipeline development and database support

The responders requested support for Microsoft Azure and Google Cloud Platform besides Amazon Web Services, as well as on-premise and managed deployments (SaaS, Figure 8). While most responders prefer to manage their pipelines individually, a self-managed environment with all setup and operations is attractive for some projects (independent of the actual pipeline implementation). Besides the runtime environment, support for the target RDF database is the most significant topic in a specific deployment scenario. Besides the existing support for Ontotext GraphDB, Stardog and Amazon Neptune are relevant in the short term, but all commercial and enterprise-grade databases are relevant (Figure 9).

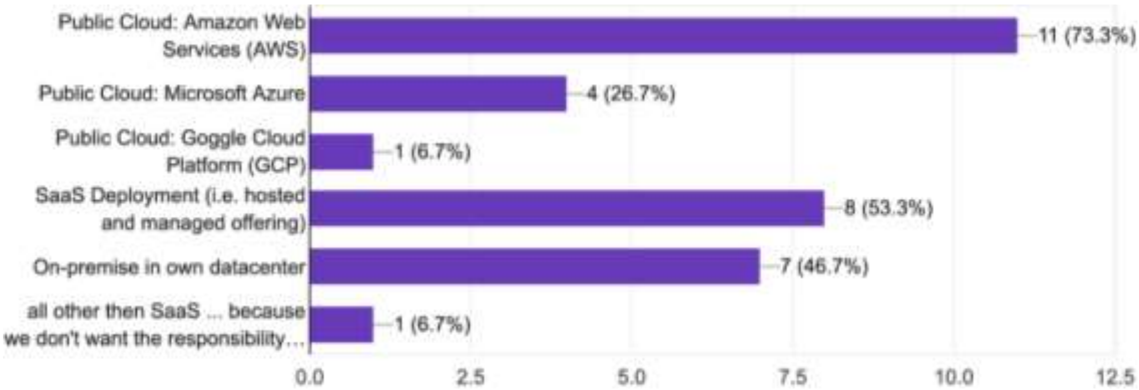

Figure 8: Graph-Inceptor ETL pipeline relevant hosting infrastructures.

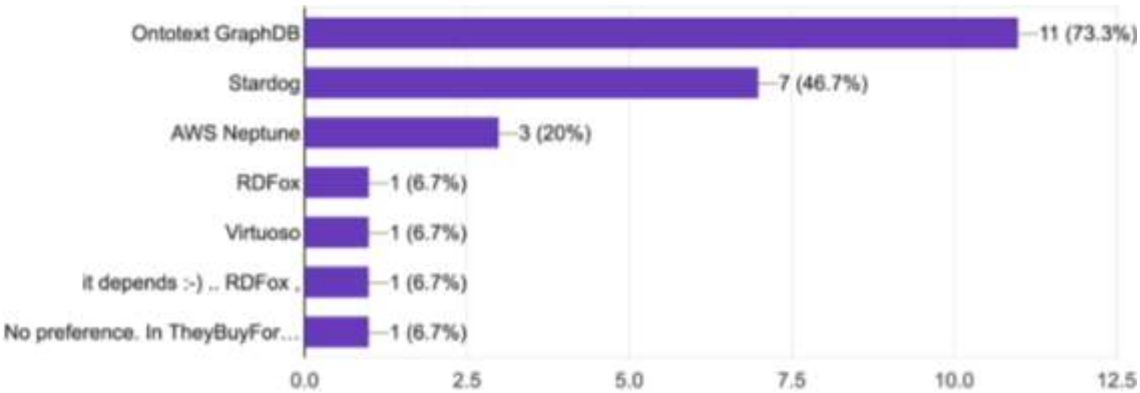

Figure 9: Graph-Inceptor ETL pipeline relevant RDF databases.

KG database update

Periodically recreating the full KG from scratch is a viable option for about half of the projects, while others require incremental updates (Figure 10). However, the duration of a single pipeline run to create the complete KG from scratch takes anywhere from a few hours to a few days, depending on the data volume. Thus, support for incremental updates is crucial for most projects, with varying requirements about update frequency, amount, and guarantees for the time lag between source data updates and database availability. The update frequency ranges from daily to monthly (Figure 11), reflected in the database within a day (Figure 12). Tracking changes and handling proper updates in the RDF database, such as deleting changed or obsolete data before adding new ones, depends on the use case.

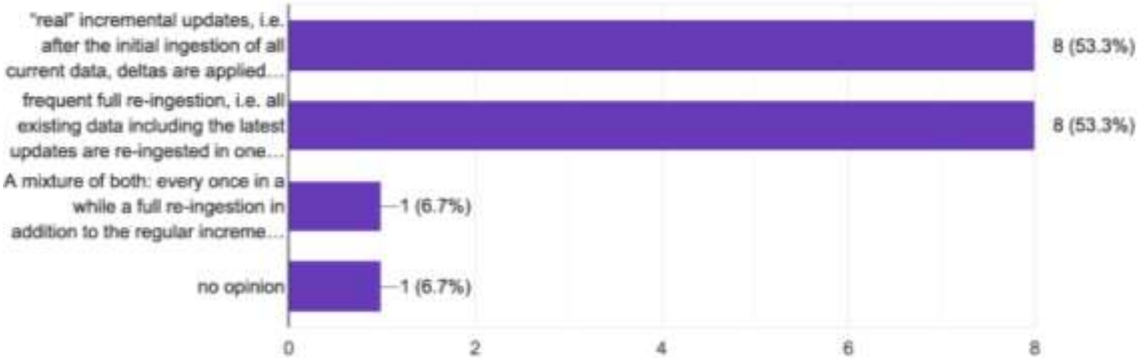

Figure 10: Graph-Inceptor ETL pipeline incremental KG update support.

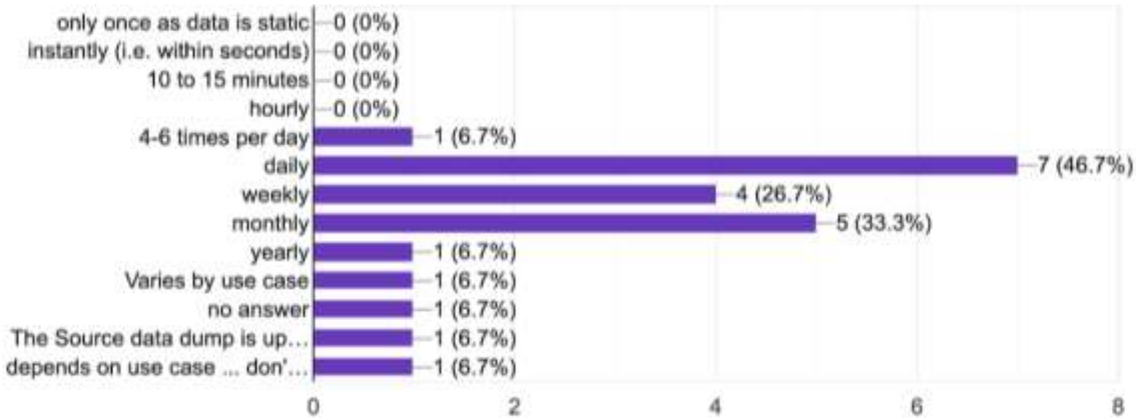

Figure 11: Graph-Inceptor ETL pipeline KG database update frequency.

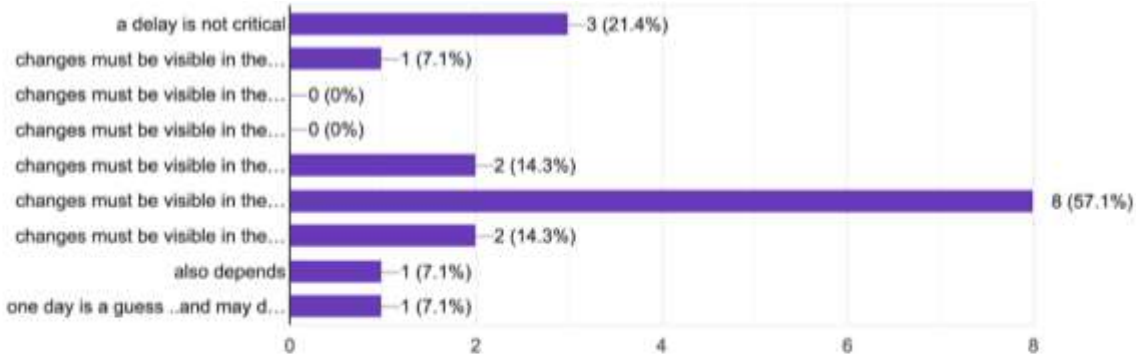

Figure 12: Update visibility in the KG database.

Compliance

Besides technical aspects, other barriers, such as compliance, hinder ETL pipeline adoption (Figure 13). Better documentation and examples and additional tooling, e.g., for authoring or visualization of mappings, can facilitate adoption.

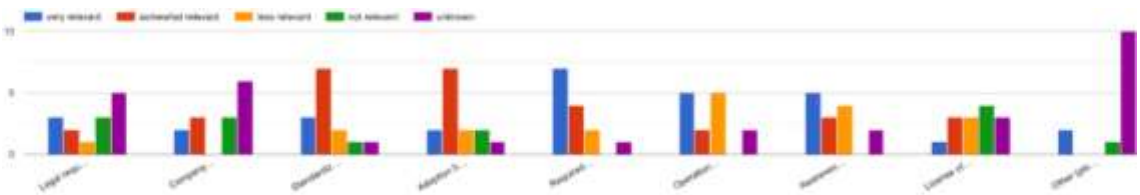

Figure 13: Barriers hindering the ETL pipeline adoption.

3.1.2 GraphMa

The beta testing survey for the GraphMa component of the Graph-Inceptor tool yielded four responses, with one participant from each use case partner as the leading developer. The beta-tester used GraphMa in four mandatory scenarios and one optional. For each scenario, the beta tester answered questions about functionality, user experience, bugs, and performance. Finally, the beta tester could also provide additional comments.

Table 3: Graph-Inceptor beta-testing GraphMa feedback summary.

Beta-testing campaign	Details
Software component	Graph-Inceptor GraphMa
Participants	4 use case partners
Topics	Functionality, user experience, bugs, performance.
Feedback summary	• Good beta testing design (5 scenarios ranging from easy to difficult; • More complex scenarios not completed by all beta-testers; • Easy running of predefined graph pipelines, not requiring programming skills; • Building custom graph pipelines requiring basic programming skills; • Sufficient support for data sources, formats, and operators; • Graph pipelines in a sequential, useful, and intuitive API; • Difficulty in enhancing GraphMa with custom operators requiring advanced programming skills.

Predefined graph pipelines

Table 4 and Figure 14 summarise the feeback of executing three predefined graph pipelines.

Table 4: Predefined graph pipeline experience.

Metric	Feedback summary
Functionality	Beta testers appreciated the packaging pipeline steps for distributed deployment and had no suggestions for improvement.
User experience	Two testers faced difficulties setting up the Java environment for running GraphMa.
Bugs	• 1 beta tester reported a bug in the build file; • 1 beta tester suggested improvements to the documentation.
Performance	The performance of the predefined graph pipelines was satisfactory for all responders.

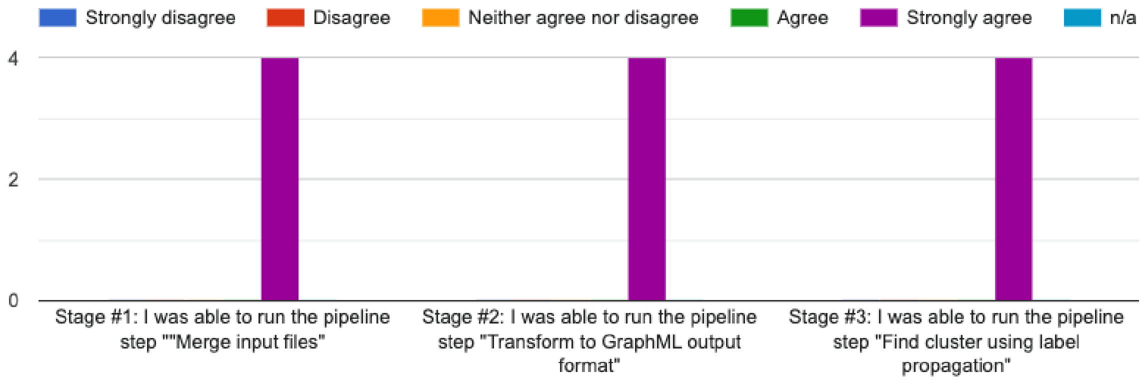

Figure 14: Precompiled graph pipeline execution experience.

Graph pipeline composition and execution

Beta testers evaluated and ranked the functionality of GraphMa concerning support for data sources and formats, coverage of operators, and use case requirements, as summarised in Table 5 and Figure 15.

Table 5: Graph pipeline composition and execution experience.

Metric	Feedback summary
Functionality	One beta-tester commented on the need for better documentation and defining the pipeline steps in the correct order.
User experience	Two beta testers lacked Java experience but could still complete the scenario tasks following the documentation and examples.
Bugs	No beta-tester reported any bugs.
Performance	The graph pipeline composition and execution performance were satisfactory to all testers.

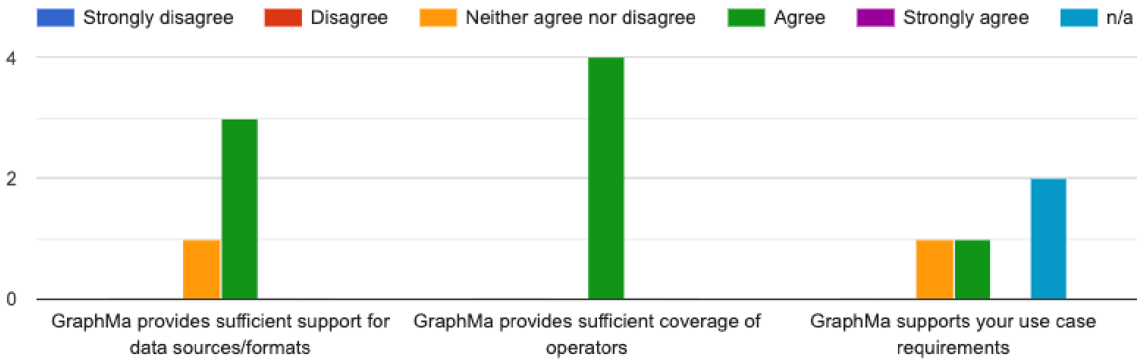

Figure 15: GraphMa graph pipeline composition and execution support.

Custom graph operator implementation

The ease of extending GraphMa with custom graph operators largely depends on Java programming skills and experience, summarised in Table 6 and Figure 16.

Table 6: Custom graph operator implementation feedback.

Metric	Feedback summary
Functionality	Beta testers also provided feedback on missing functionality concerning support for financial data operators, graph neural network inference related to the use cases, and more general operators from other libraries.
User experience	• Inheriting from an abstract base class eases extending GraphMa with custom operators; • Two beta testers reported difficulties due to limited Java experience.
Bugs	No beta-tester reported any bugs.
Performance	No beta-tester reported any performance issues.

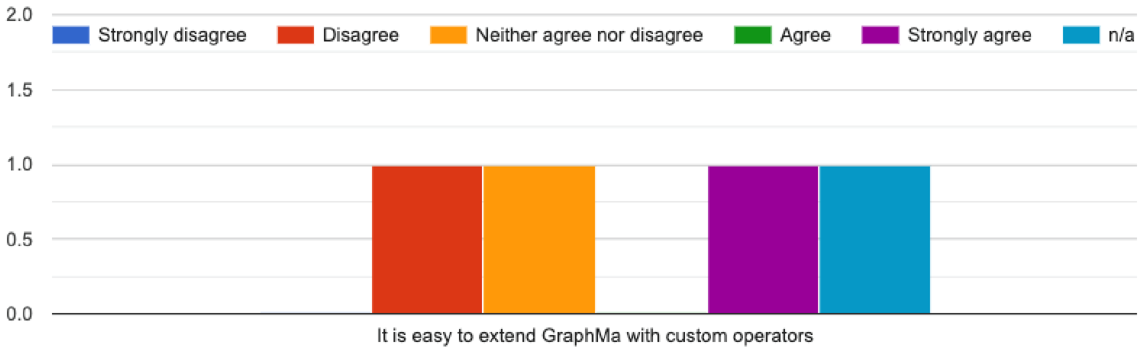

Figure 16: Custom graph operator implementation experience.

Custom traverser operator implementation

The ease of extending GraphMa with custom graph traverser operators largely depends on Java programming skills and experience, summarised in Table 7 and Figure 17.

Table 7: Custom traverser operator implementation experience.

Metric	Feedback summary
Functionality	Two testers agreed on traversers as an appropriate abstraction for graph data sources.
User experience	Two beta testers reported difficulties due to limited Java experience.
Bugs	One beta-tester reported a bug in one source file.
Performance	No beta-tester reported any performance issues.

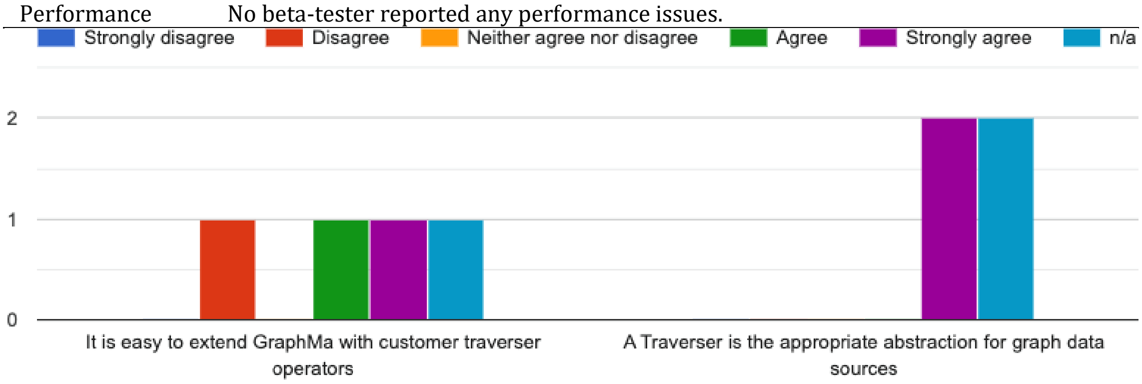

Figure 17: Custom traverser operator implementation experience.

Custom data sources and graph processing pipelines

Only one participant completed the optional scenario of customising data sources and graph processing pipelines that require advanced Java programming experience (see Table 8).

Table 8: Custom data sources and graph processing pipeline experience.

Metric	Feedback summary
Functionality	One beta tester succeeded in a basic test on financial data.
User experience	• One beta tester failed to succeed due to limited Java experience; • The README file misses information or clarity.
Bugs	Weighted graphs do not support negative weights.
Performance	No beta-tester reported any performance issues.

3.2 GRAPH-SCRUTINIZER

Graph-Scrutinizer conducted two beta-testing surveys, which yielded responses from industrial researchers and related to the Graph-Massivizer project. The questions relate to three main topics: functionality, performance, and user experience.

3.2.1 ts2g2 Library

The ts2g2 library received answers from six beta testers, summarised in Table 9.

Table 9: Graph-Scrutinizer ts2g2 library beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Scrutiniser ts2g
Participants	6 industrial and research experts
Topics	Functionality, performance, user experience.
Feedback summary	• Limited instructions; • No apparent execution bugs; • Unclear result interpretability.

Functionality

Figure 18 shows that the ts2g2 library received good evaluations except for one participant, who did not understand the documentation. A helpful suggestion was to give the time series a colour gradient identical to the respective nodes to visualise their partial relation better. There is a need for better documentation, especially its goal since some comments were out of the library scope (e.g., interactive graph visualisations, custom coarse-grained time series granularity for compact graph generation, and online data retrieval).

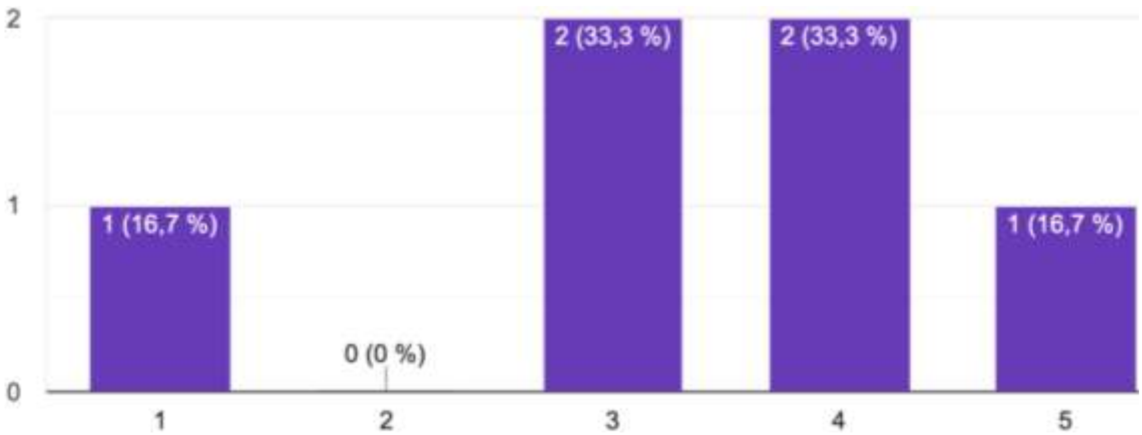

Figure 18: Graph-Scrutinizer ts2g2 library functionality evaluation (1: poor, 5: excellent).

Performance

Figure 19 and Figure 20 show that participants mostly worked with time series that resulted in 500-node graphs in their beta-testing scenarios and evaluated them well, except for one participant who did not understand the documentation. The participants did not prefer a particular graph to a time series strategy, considering them functionally appropriate, effective, and bug-free.

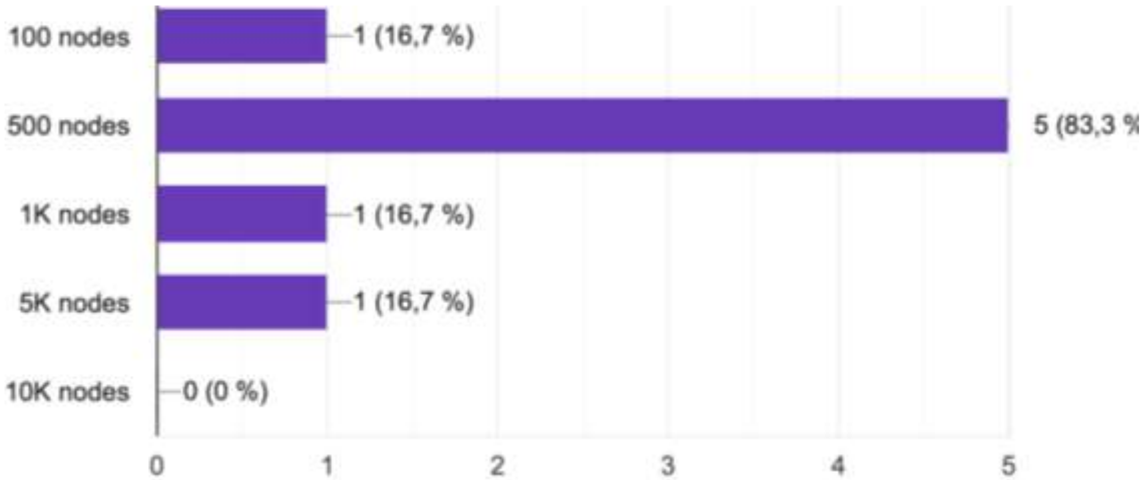

Figure 19: Graph-Scrutinizer ts2g2 library times series length to graph size evaluation.

15

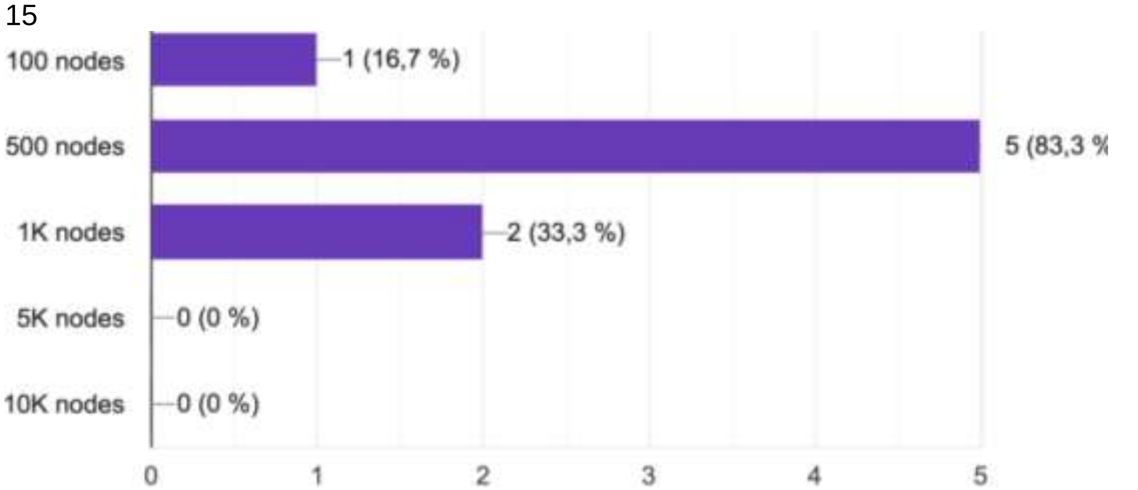

Figure 20: Graph-Scrutinizer ts2g2 library graph sizes to times series lengths evaluation.

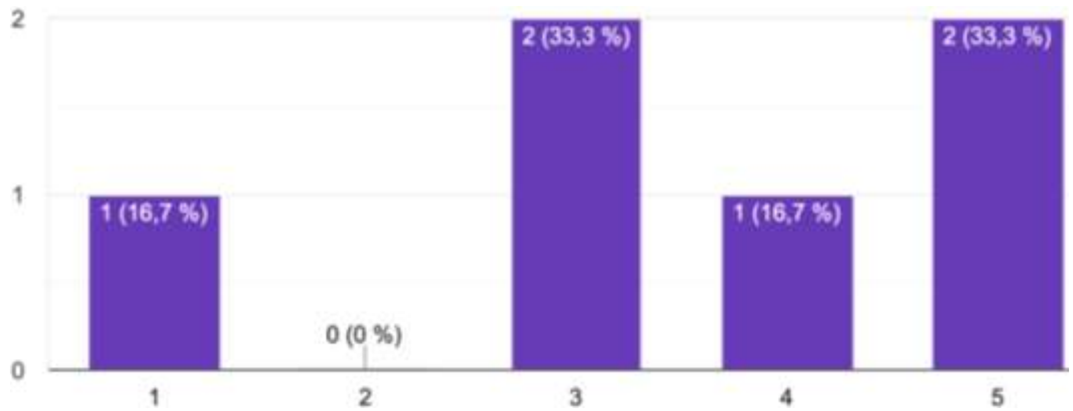

Figure 21: Graph-Scrutinizer ts2g2 library generated time series evaluation (1: poor; 5: excellent).

User experience

Suggestions for enhancing the ts2g2 library documentation comprise:

- A step-by-step tutorial with clear scenarios and execution goals;
- Detailed explanation of the graph semantics, especially its nodes and edges, difficult to relate to time series for novice users;
- Detailed explanation of the graph features relevant to each time series to graph conversion method producing very different graphs;
- Easier scenario instantiation;
- Unclear problem size and implementation testing scenarios;
- Dynamic graph generation for long time series and unnatural or unpromising plotting of dense graphs;
- Several bugs, comprising plotting results, duplicate script references and missing imported or implemented modules.

3.2.2 Graph-Scrutinizer

The Graph-Scrutinizer component received answers from three beta-testing participants, summarised in Table 10.

Table 10: Graph-Scrutinizer beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Scrutinizer
Participants	6 industrial and research experts
Topics	Functionality, performance, user experience.
Feedback summary	• Complicated setup with multiple programs but relatively easy to run; • Missing programs' feedback (running silently or freezing?); • Limited instructions for using the REST API and unclear expected results.

Performance

All participants tested both tiny and small graph sizes and confirmed nearly instant to short (few seconds) execution times (Figure 22).

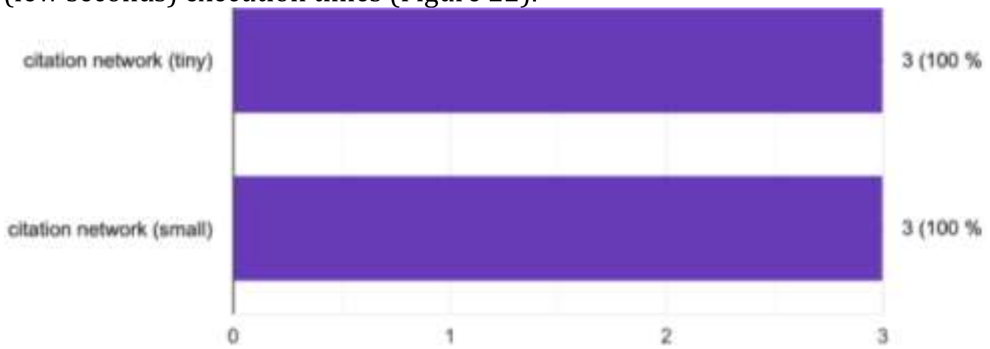

Figure 22: Graph-Scrutinizer tested graph sizes.

User experience

One participant failed to provide the input graph to the example and reported the indefinite running of the `consumer-betweenness-centrality.py` script, indicating the need for better documentation. All participants indicated difficulties running the examples (Figure 23) due to the lack of clarity (Figure 24). Other detailed comments include:

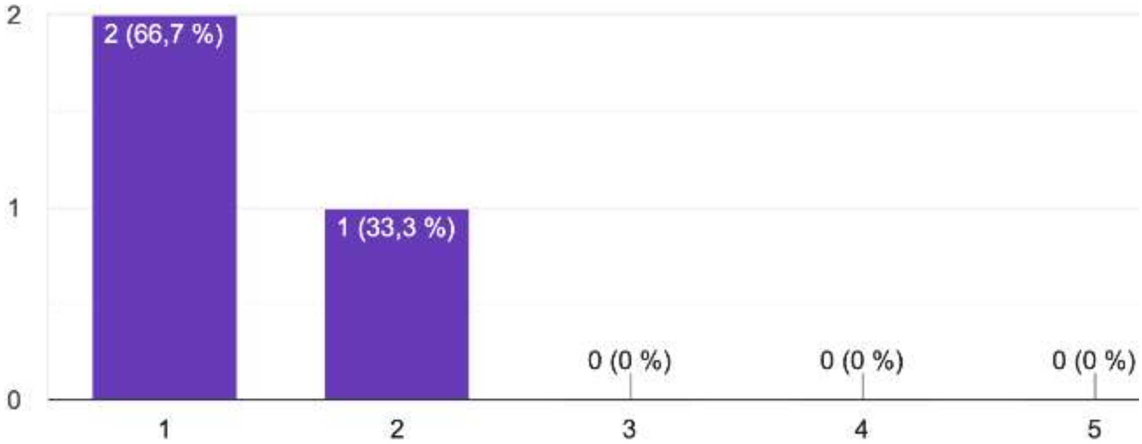

Figure 23: Graph-Scrutinizer example execution difficulty (1: very difficult; 5: very easy).

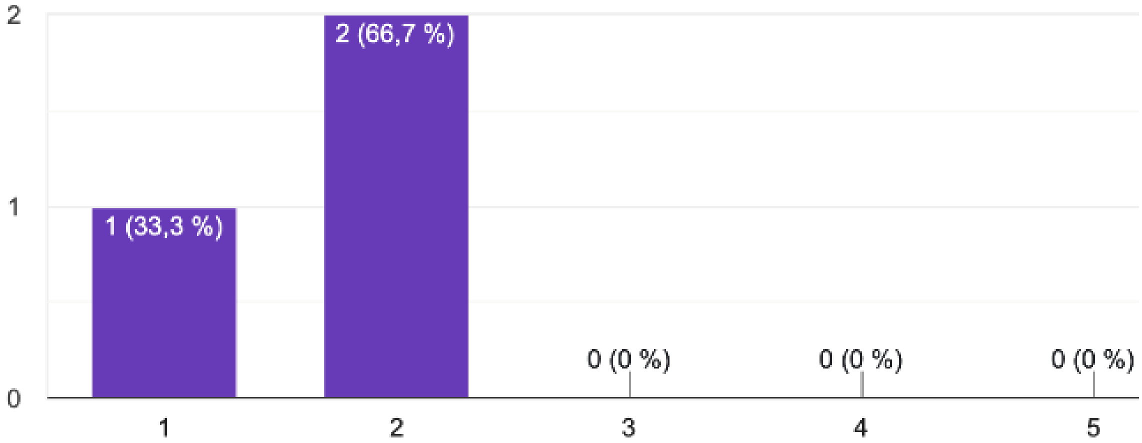

Figure 24: Graph-Scrutinizer example clarity (1: unclear; 5: crystal clear).

- Unclear multiple steps requiring a to open another terminal (e.g., `tmux` session) for keeping the previous programs running;
- No interactive progress report in `consumer-betweenness-centrality.py` script ;
- Incomplete Python environment package installation guide;
- Unclear references to multiple README.md files;
- Unclear beginner use of FastAPI;
- Missing default location for downloading files during POST request in FastAPI;
- Unclear successful requests, returning code 200;
- No Microsoft Windows documentation, such as environment variables and path syntax;
- Improved result presentation like score or visualisation.

3.3 GRAPH-OPTIMIZER

The Graph-Optimizer beta-testing follows four steps, covering BGO workload specification (Table 11), BGO calibration (Table 12), prediction (Table 13), and graph properties (Table 14). Overall, the tool functions as expected internally, but the user interface is challenging. Since Graph-Optimizer is not a front-end tool, clarifying the outputs communicated to the users will take priority, and the rest will receive partial importance.

Table 11: Graph-Optimizer BGO workflow specification feedback.

Beta-testing campaign	Details
Software component	Graph-Optimizer BGO workflow specification
Participants	3 students and 2 staff colleagues (3 project-external and 2 outside UTW).
Topics	BGO workflow format, workflow creation according to desired functionality.
Feedback summary	• Easy to read Workflow in the proposed format (23 / 25); • No functional requirements included in the workflow: feature or bug?; • Clear and straightforward workflow creation from available BGOs (23 / 25); • Cumbersome JSON interface (graphical or text-based necessary).

Table 12: Graph-Optimizer hardware configuration and micro-benchmarking feedback.

Beta-testing campaign	Details
Software component	Graph-Optimizer hardware configuration and micro-benchmarking
Participants	3 students and 2 staff colleagues (3 project-external and 2 outside UTW).
Topics	Hardware specification; micro-benchmarking; results' readability and interpretability.
Feedback summary	• Easy to run microbenchmarks but unclear use and need of results (19 / 25); • Hard to find machine specifications (no autodetect) and user input translation; • Unclear goal of changing hardware configuration; • Unclear outcome of complex configurations (1 / 5); • Missing power micro-benchmarking; • Compiler incompatibility errors.

Table 13: Graph-Optimiser prediction feedback.

Beta-testing campaign	Details
Software component	Graph-Optimiser prediction
Participants	3 students and 2 staff colleagues (3 project-external and 2 outside UTW).
Topics	Prediction run, results' readability and interpretability, hardware adaptation.
Feedback summary	• Easy out-of-the-box prediction run (25 / 25); • Clear results but missing units (19 / 25); • Per-BGO, graph and subgraph predictions; • Missing communication prediction, saving prediction and comparison; • No adaptation for power parameters; • Unclear flow of the demo and questionnaire.

Table 14: Graph-Optimiser graph properties feedback.

Beta-testing campaign	Details
Software component	Graph-Optimizer graph properties
Participants	3 students and 2 staff colleagues (3 project-external and 2 outside UTW).
Topics	Adaptation to graph properties; Results readability.
Feedback summary	• Visible impact of graph characteristics impact (25 / 25); • Limited interface for graph update (easier to upload a new graph); • Difficult correlation from basic properties to actual graph features; • Unclear flow of demo and questionnaire.

3.4 GRAPH-GREENIFIER

The Graph-Scrutinizer component received answers from seven beta-testing participants, comprising four students and three staff members. Their task was to run a demonstration of the Graph-Greenifier tool containing six parts grouped into two sections, summarised in Table 15 and Table 16.

- **Simulation** focuses on running an experiment, explaining all input files and asking the user to create new ones;
- **Analysis** focuses on the generated output files, shows several examples, and asks users to implement new ways to analyse and visualise the data.

Table 15: Graph-Greenifier simulation setup beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Greenifier simulation setup
Participants	4 students, 3 academic staff members.
Topics	Setup, topologies, runtime scenarios.
Feedback summary	• Easy to define data centre hardware topology; • Cumbersome JSON format and unit specification; • Unstraightforward adding of new components and traces; • Missing exampling for creating custom scenarios.

Table 16: Graph-Greenifier beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Greenifier
Participants	4 students, 3 academic staff members.
Topics	Raw output, aggregation, visualisation.
Feedback summary	• Hard to understand terminology and metrics; • Hard to identify and compare multiple scenarios; • Easy creation of plots from data; • Clear relation between system performance metrics and carbon emissions.

Topology

Topology files define the hardware of a data centre required to run an experiment in JSON format. Most users understood the formal explanation and implementation (see Figure 25). Based on this knowledge, all of them could determine the hardware of another data centre based on the provided topology file and vice versa. However, one user needed slight help (see Figure 26). Suggestions to improve the topology file include improved documentation and error-prone raw JSON as input requiring higher-level abstractions.

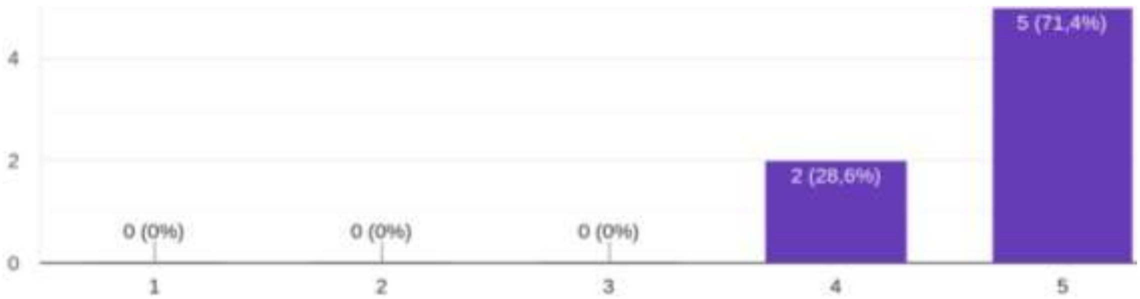

Figure 25: Graph-Greenifier topology format and implementation understanding.

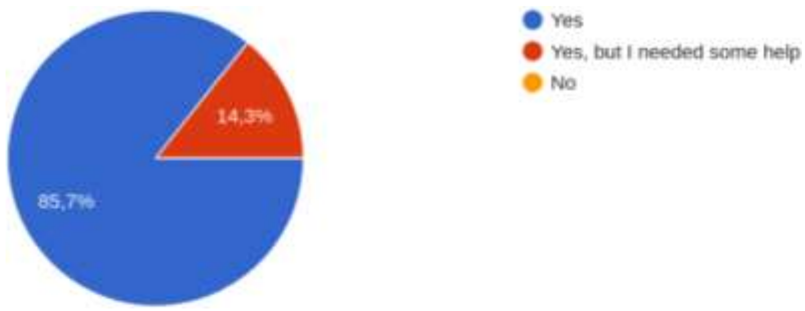

Figure 26: Graph-Greenifier medium topology file creation experience.

Scenario

A scenario file defines a simulation run, again using a JSON file. All users understood the scenario files well based on the examples (Figure 27) and could create a custom one for running their own experience. However, the experience was slightly more complex than the topology files (Figure 28). Despite the positive experience, most users see documentation improvements as possible.

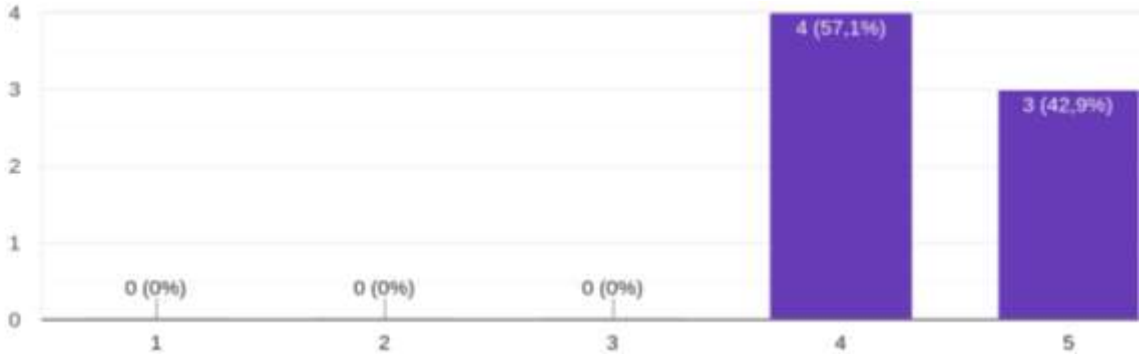

Figure 27: Graph-Greenifier scenario understanding.

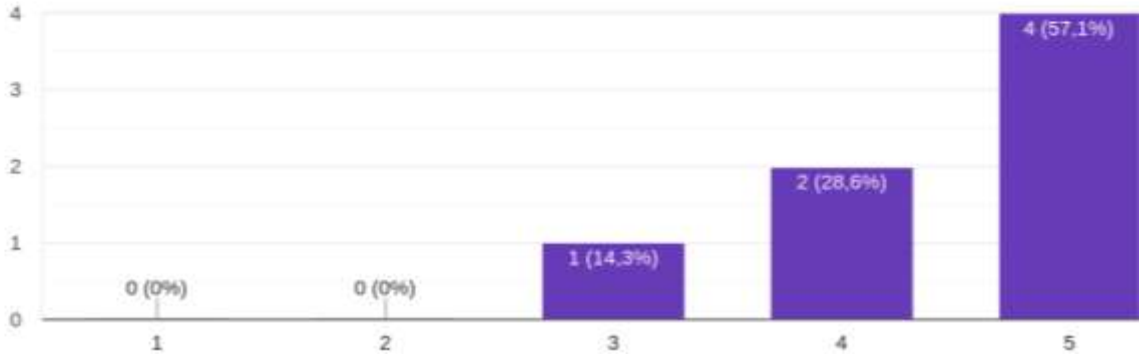

Figure 28: Graph-Greenifier custom scenario creation experience.

Execution

All participants could successfully execute the simulator on the provided input files with a runtime between 5 and 8 seconds, depending on the underlying system.

Raw output analysis

All participants could understand the three generated output files, host, server, and service, except one having some difficulties (Figure 29). However, two of them suggested improving the usability of the tool.

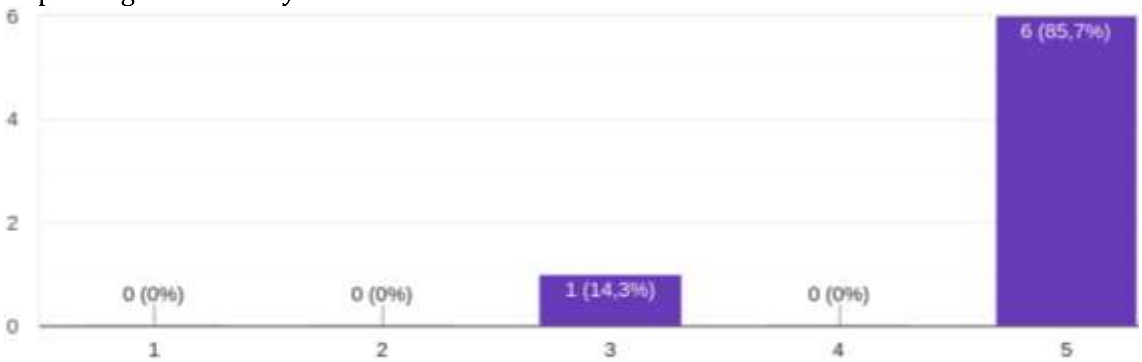

Figure 29: Graph-Greenifier output file understanding.

Aggregation

Aggregating results is a vital part of data analysis that the participants understood well (Figure 30). To test their understanding, they could successfully determine the average number of tasks waiting for a host, except for one user who needed help (Figure 31) and asked to improve the documentation. Missing functionality includes a separate carbon function, stakeholder-specific aggregation functions, and the use of standard deviation to represent fluctuations over time.

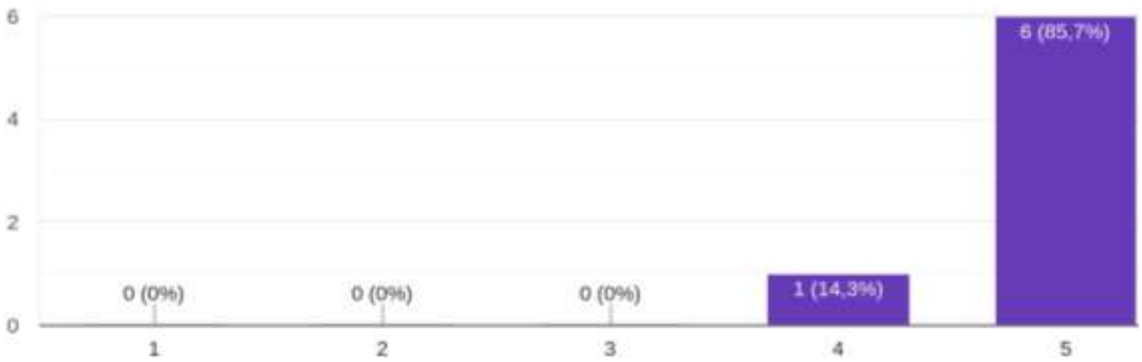

Figure 30: Graph-Greenifier data aggregation understanding.

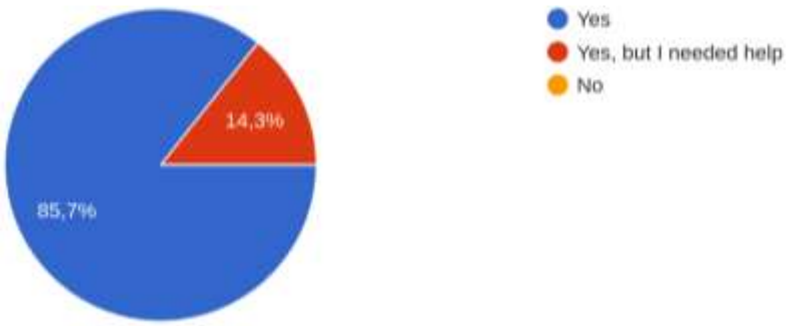

Figure 31: Graph-Greenifier average number of tasks determination experience.

Visualisation

Visualising raw output data is essential for providing new insights. All participants understood the visualisation capabilities of the Graph-Greenifier and could create new ones from the service (Figure 32) and host (Figure 33) output following the plotting examples based on different features. All users except one managed to visualise the pending tasks and carbon emissions (Figure 34) but requested more options to improve their presentations (e.g., colours, labels).

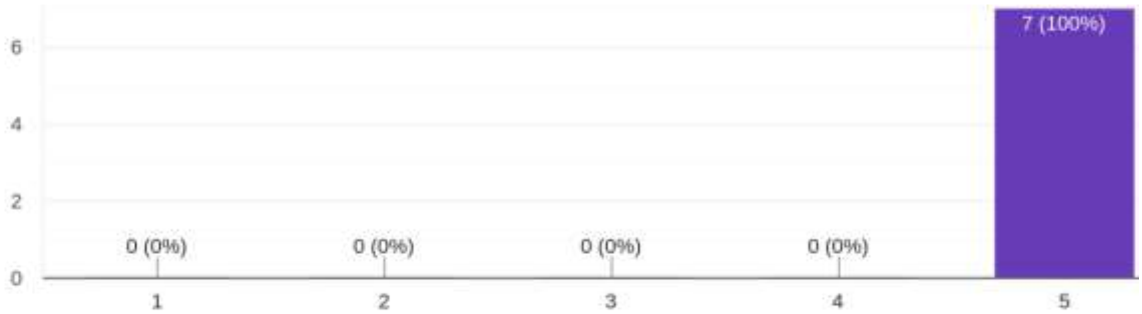

Figure 32: Graph-Greenifier custom visualisation understanding from service output files.

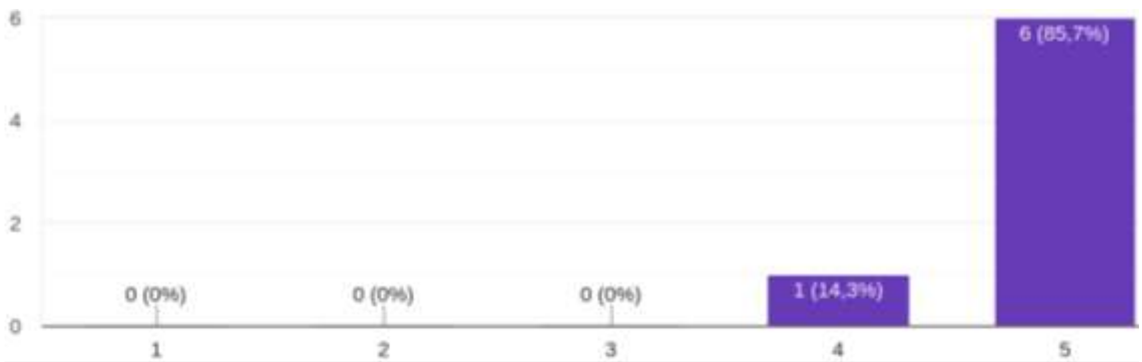

Figure 33: Graph-Greenifier custom visualisation understanding from host output files.

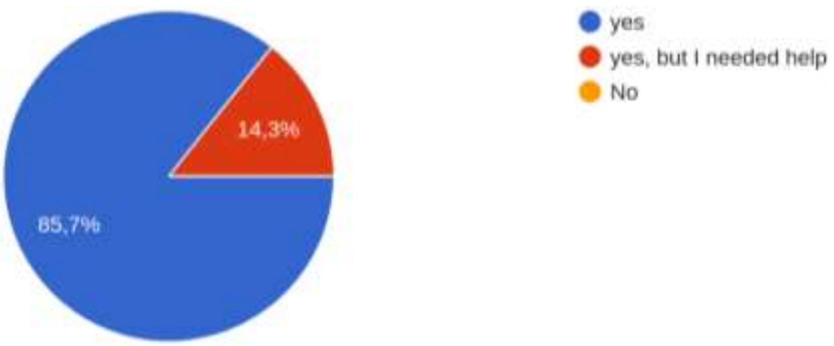

Figure 34: Graph-Greenifier visualisation of pending tasks and carbon emissions experience.

3.5 GRAPH-CHOREOGRAPHER

Graph-Choreographer received answers from ten participants at AAU, comprising three researchers and seven students. Their task was to run a demonstrator on the computing continuum, organised into two sections, summarised in Table 17 and Table 18.

- **Serverless workflow scheduling** focuses on generating and visualising optimised execution plans for input workflow functions based on specified objectives, function requirements, and infrastructure characteristics;
- **Execution engine** follows the scheduling plans, runs the workflow functions across instances on the computing continuum, and provides real-time monitoring to showcase the impact of scheduling choices.

Table 17: Graph-Choreographer workflow scheduling beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Choreographer scheduling
Participants	10 researchers and students.
Topics	Documentation, installation, scheduling, example functions, infrastructure, monitoring, template.
Feedback summary	• Clear template for inputting scheduling modules; • Single function, workflow, and workflow batch scheduling support; • Comprehensive scenarios challenging to execute without backend knowledge; • Sufficient metrics lacking adequate visualisation; • Clear scheduling policy optimising function time and cost but lacking energy.

Table 18: Graph-Choreographer execution engine beta-testing feedback summary.

Beta-testing campaign	Details
Software component	Graph-Choreographer execution engine
Participants	10 researchers and students.
Topics	Documentation, installation, example workflows, infrastructure, monitoring, orchestration platform, serverless providers.
Feedback summary	• Numerous instances available for execution scenarios; • Lack of a user interface or dashboard for scheduling output execution; • Sufficient metrics lacking adequate visualisation; • No execution support across multiple regions.

Documentation quality and clarity

The documentation on input workflows, scheduling objectives, infrastructure setup, and execution received positive feedback on its foundational descriptions, references, and complementary publications (Figure 35). However, several users requested exact command examples and clearer summaries of essential technologies, while others suggested video clips to visualise setups. A few testers had to consult additional publications to understand certain concepts, and they recommended their consolidation in a summarised folder.

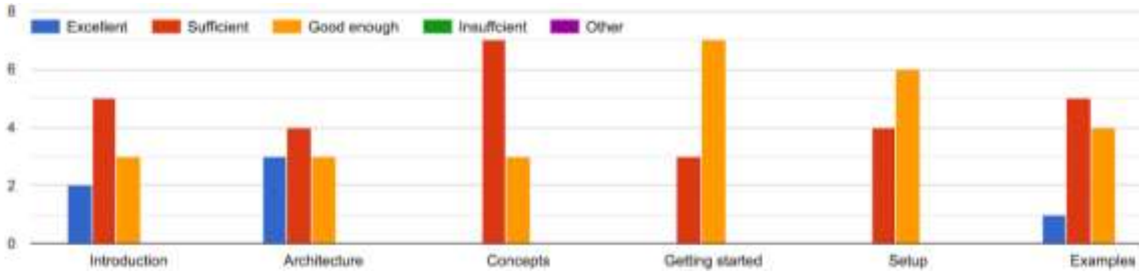

Figure 35: Graph-Choreographer documentation quality and clarity experience.

Installation

Installing Graph-Choreographer modules was challenging for most users, indicating moderate to relative difficulty (Figure 36). However, beta testers found the provided templates highly effective for implementing specific workflows and testbed environments, with most users rating them as exceptionally useful (Figure 37). The templates supported diverse testing needs, proving valuable for workflow setup and integration. While assistance from the tool developer clarified steps, users expressed a need for more detailed instructions to improve accessibility. Specific feedback included requests for step-by-step commands for installation and subsequent tasks like running, benchmarking, monitoring, and visualisation, such as a short video tutorial.

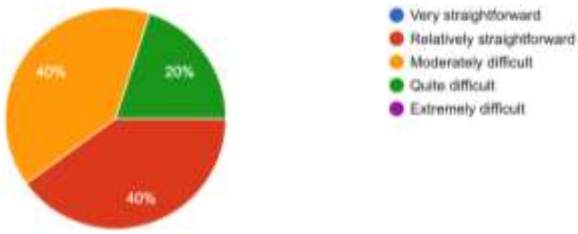

Figure 36: Graph-Choreographer installation experience.

Figure 37: Graph-Choreographer installation template usefulness.

Workflow scheduling and execution

The sample workflow allowed testers to evaluate Graph-Choreographer in scheduling and coordinating complex distributed tasks (Figure 38). The scheduling module offered satisfactory insights into function prioritisation and computing continuum resource allocation using completion time and economic cost metrics (Figure 39). However, users expressed interest in expanding the focus to energy metrics and the impact on resource use and environmental footprint. Most users preferred running single workflows rather than batches (Figure 40). The execution module's functionality and adaptability for testers' workflows received favourable ratings (Figure 41). Despite some challenges with complex modules, users suggested more visualisation and flexible templates, and most will likely recommend Graph-Choreographer for its workflow management capabilities (Figure 42).

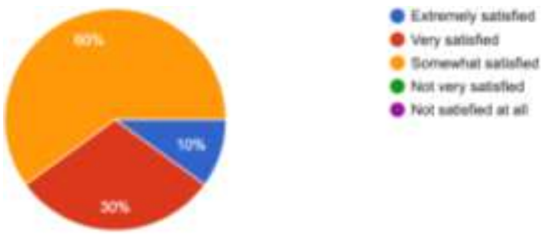

Figure 38: Graph-Choreographer sample workflow relevance and experience.

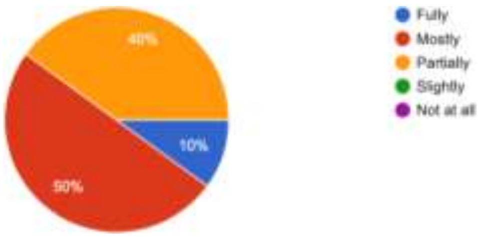

Figure 39: Graph-Choreographer workflow scheduling performance satisfaction.

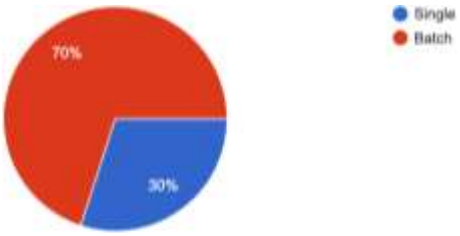

Figure 40: Graph-Choreographer execution mode preferences.

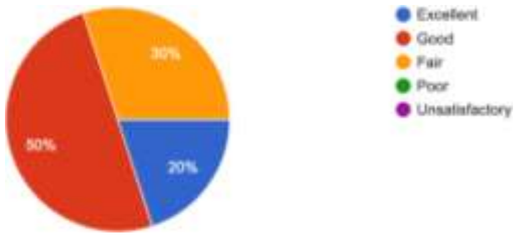

Figure 41: Graph-Choreographer execution module user experience.

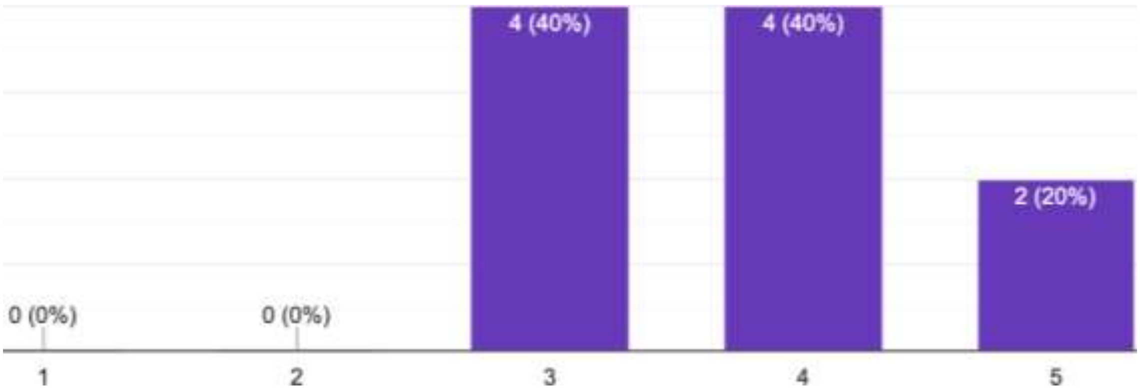

Figure 42: Graph-Choreographer recommendation intent (1: do not recommend, 5: highly recommend).

3.6 BETA-TESTING IMPLICATIONS

Table 19 summarises the implications of beta-testing campaigns on the development timeline of the five tools concerning resource availability and eventual risk mitigation.

Table 19: Implications of beta-testing campaigns tool development.

Tool	Resource Availability	Implementation Timeline	Risk Mitigation
Graph-Inceptor	Sufficient resources.	Missing use case functionality confirming the requirements.	Performance issues uncovered by benchmarking.
Graph-Scrutinizer	More resources necessary for documentation.	Delayed functionality implementation after its documentation.	• Limited usefulness and reusability due to lack of documentation; • Tighter integration with Graph-Inceptor.
Graph-Optimizer	Sufficient resources.	Possible delays due to complex caching mechanism.	Design of a basic tool-specific cache.
Graph-Greenifier	Sufficient resources.	Improved documentation already included in the timeline.	Improvement of documentation and user interaction.
Graph-Choreographer	More infrastructure necessary for scalability evaluation.	Additional effort for scheduling for energy and carbon footprint.	Expanding hardware testing may uncover performance bottlenecks or incompatibilities.

4 REQUIREMENTS EVOLUTION

This section presents an update of the Graph-Massivizer platform requirements and its constituent tools as an evolution of the first elicitation formulated in deliverable D2.1 following the beta-testing feedback results presented in Section 3.

4.1 GRAPH-MASSIVIZER PLATFORM

4.1.1 Description and Mapping

Identifier	GM
Title	Graph-Massivizer platform
Source	MPH, State-of-the-art analysis; Focus groups; Stakeholder interviews; User stories.
Tags	AI; ML; Data science; Data analysis; KG; KG Engineer, Semantics.
Description	Graph-Massivizer platform integrating the project tools and executing the four UCs.
Responsible	MPH
Stakeholders	AI expert, data scientist, KG engineer.
Preconditions	• Graph-Massivizer tool releases; • UCs releases providing the required input data; • Metaphactory platform ready for integration.
Success conditions	• All UCs run smoothly and provide the expected results;
Failure conditions	• At least one UC fails to run correctly;

4.1.2 Graph-Massivizer Platform Documentation

Requirement identifier	GM-1
Requirement name	Graph-Massivizer shared data model
Priority	High
Description	Represent and link data externally accessible in each tool's shared ontology or explicit model.
Precondition	–
Source	Graph-Massivizer architecture
Category	Functional; System; User.

Requirement identifier	GM-2
Requirement name	Graph-Massivizer real-time processing tools
Priority	Very high
Description	Process extreme data streams represented as massive graphs in real-time.
Precondition	Unblocking data processing tasks
Source	Graph-Massivizer architecture
Category	Non-functional.

Requirement identifier	GM-3
Requirement name	Graph-Massivizer application integration
Priority	Very high
Description	Deploy the tools and toolkit according to the system architecture design, accomplished directly by running code or providing an API or a shared library dependent on the specific needs and functionalities.
Precondition	Toolkit architecture design and implementation
Source	Graph-Massivizer architecture
Category	Functional; Design; System; User.

Requirement identifier	GM-4
Requirement name	Graph-Massivizer internal RDF-star format for u
Priority	High
Description	Interface the Graph-Massivizer tools with the toolkit in a standardised RDF format via the APIs and share compatible data across the toolkit.
Precondition	GM-1
Source	Graph-Massivizer architecture
Category	Functional; Design; System.

Requirement identifier	GM-5
Requirement name	Graph-Massivizer SPARQL data queries
Priority	High
Description	Offer SPARQL data queries to tool and toolkit stakeholders for requesting data on specific topics of interest; extend the integrated data model to allow access to relevant data subsets based on use case or stakeholder requirements.
Precondition	Data reachability
Source	Graph-Massivizer architecture
Category	Functional; User.

Requirement identifier	GM-6
Requirement name	Graph-Massivizer graph and data visualisation
Priority	High
Description	Support visualisations to allow stakeholders to inspect and analyse the data without parsing it raw.
Precondition	GM-1, GM-2, GM-3, GM-4, GM-5
Source	Graph-Massivizer architecture
Category	Non-functional; User.

Requirement identifier	GM-7
Requirement name	Configurable resource management
Priority	Very high
Description	Deploy the toolkit in an integrated environment for Graph-Choreographer to effectively manage BGOs.
Precondition	Graph-Inceptor, Graph-Scrutinizer, Graph-Choreographer
Source	Graph-Massivizer architecture
Category	Functional; Design; System.

4.1.3 Graph-Inceptor Requirements Documentation

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
Source	Graph-Massivizer architecture
Category	Functional; Design; System.

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
Source	Graph-Massivizer architecture
Category	Functional; Design; System.

Requirement identifier	GI-3
Requirement name	Graph generation BGO functionality
Priority	Very high
Description	Release graph generation functionality as BGO usable by the other tools, comprising loading, filtering, and storing graph data in a graph database.
Precondition	GI-1, BGO, communication model
Source	Graph-Massivizer architecture, D3.1
Category	Functional; Design; System.

Requirement identifier	GI-4
Requirement name	Graph conversion BGO functionality
Priority	Very high
Description	Release graph conversion functionality as BGO usable by other tools, particularly Graph-Scrutinizer, comprising data to graph, interface to graph, and graph to file.
Precondition	GI-2, BGO, communication model
Source	Graph-Massivizer architecture, D3.1
Category	Functional; Design; System.

4.1.4 Graph-Scrutinizer Requirements Documentation

Requirement identifier	GS-1
Requirement name	Multiple graph view 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
Source	Graph-Massivizer architecture
Category	Non-functional; User.

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
Source	Graph-Massivizer architecture
Category	Functional; Design; System.

Requirement identifier	GS-3
Requirement name	Reduced graph size preserving relevant properties
Priority	High
Description	Support different subgraph views for faster and more focused analytics on particular data of interest to a user, generated using graph sampling and summarization methods.
Precondition	Graph-Inceptor, Graph-Scrutinizer
Source	Graph-Massivizer architecture
Category	Non-Functional; User.

Requirement identifier	GS-4
Requirement name	Heuristic and graph machine learning analysis
Priority	High
Description	Support multiple primitives for heuristic and machine learning approaches on graphs at scale.
Precondition	Graph-Inceptor, Graph-Scrutinizer
Source	Graph-Massivizer architecture
Category	Non-Functional; User.

4.1.5 Graph-Optimizer Requirements Documentation

Requirement identifier	GO-1
Requirement name	Hardware agnosticism
Priority	High
Description	Support agnostic operations across different hardware types and infrastructures with consistent behaviour across diverse contexts and UCs.
Precondition	Graph-Optimizer
Source	Graph-Massivizer architecture
Category	Non-functional; Design; System.

Requirement identifier	GO-2
Requirement name	Detailed BGO and subgraph predictions
Priority	Moderate
Description	Provide predictions per BGO and per subgraph alongside the original best end-to-end performance.
Precondition	BGO, communication models
Source	Graph-Massivizer architecture
Category	Functional; User interface.

Requirement identifier	GO-3
Requirement name	Provide back-up prediction via benchmarking
Priority	High
Description	Use a sample of the graph and the target platform as a performance model in case of unfeasible analytical models, further inferred by scaling up the predictions and using a simple power-time model for energy.
Precondition	Hardware; Benchmarking data
Source	Graph-Massivizer architecture
Category	Functional; System

4.1.6 Graph-Greenifier Requirements Documentation

Requirement identifier	GG-1
Requirement name	Sustainability
Priority	High
Description	Support comparison of the sustainability of graph processing compared to past configurations or other related platforms and comparable green solutions.
Precondition	Graph-Greenifier
Source	Graph-Massivizer architecture
Category	Non-functional; System; User.

Requirement identifier	GG-2
Requirement name	Multi-faceted sustainability nature
Priority	High
Description	Improve sustainability of graph processing infrastructure by using periods of high renewable energy, more efficient hardware, and efficient algorithms using multiple comparative variants in experiments with different configurations.
Precondition	Graph-Greenifier
Source	Graph-Massivizer architecture
Category	Non-functional; System; User.

Requirement identifier	GG-3
Requirement name	Multi-faceted sustainability nature
Priority	High
Description	Improve sustainability of graph processing infrastructure by using periods of high renewable energy, more efficient hardware, and efficient algorithms using multiple comparative variants in experiments with different configurations.
Precondition	Graph-Greenifier
Source	Graph-Massivizer architecture
Category	Non-functional; System; User.

Requirement identifier	GG-4
Requirement name	Support scaling graph workloads to different infrastructure sizes
Priority	Moderate
Description	Execute, port, scale and validate graph processing workloads on infrastructures of different sizes, from a single computer to a thousand-machine cluster, using BGO descriptions and communication models.
Precondition	Graph-Greenifier
Source	Graph-Massivizer architecture
Category	Functional; System; User.

4.1.7 Graph-Choreographer Requirements Documentation

Requirement identifier	GC-1
Requirement name	Infrastructure types selection
Priority	High
Description	Select an optimized infrastructure type suited to graph analytics and their computational, storage, energy, and other cost requirements.
Precondition	Graph-Choreographer; Graph-Scrutinizer
Source	Graph-Massivizer architecture
Category	Functional.

Requirement identifier	GC-2
Requirement name	Cybersecurity methods
Priority	Very high
Description	Indicate the cybersecurity methods employed and alert in case of any identified security issues or threats.
Precondition	Graph-Choreographer
Source	Graph-Massivizer architecture
Category	Non-Functional; Design; System; User.

Requirement identifier	GC-3
Requirement name	Concurrent BGO workflow runtime scheduling
Priority	Very high
Description	Enhance Graph-Choreographer to schedule and execute multiple concurrent BGOs and workflow batches, considering runtime selection with performance, priority and sustainability metrics.
Precondition	Graph-Choreographer
Source	Graph-Massivizer architecture
Category	Functional; System.

Requirement identifier	GC-4
Requirement name	Graph-Massivizer toolkit orchestration
Priority	Moderate
Description	Orchestrate infrastructure storage, bandwidth, and computational resources to run Graph-Massivizer tools alongside scheduling and executing BGOs and workflows.
Precondition	Graph-Choreographer
Source	Graph-Massivizer architecture
Category	Functional; System.

4.2 USE CASES

New state-of-the-art market analysis and beta-testing results triggered minor evolution in the four use case requirements. This section includes only updates to the original UCs' description, mapping and requirements documentation presented in D2.1 to avoid redundant presentation.

4.2.1 Green and Sustainable Finance

Description and mapping

Table 20: UC-1 details: green and sustainable finance.

Identifier	UC-1
Title	Green and sustainable finance
Source	PER; State-of-the-art analysis; Focus group; Stakeholder interviews; User stories.
Tags	Financial historical data, stocks, commodities futures, synthetic data, KG.
Description	• Graph-Massivizer platform ○ Map the historical financial data structure into a financial KG; ○ Generate synthetic data using the financial KG, preserving the original historic statistical features (time series of USA and EU stocks and commodities options); ○ Interpolate missing data (e.g. gaps, errors) and find patterns and correlations by using ML-based inference and reasoning methods; • Peracton platform ○ Simulate financial investments and trading and recommend the greenest opportunities.
Responsible	PER
Stakeholders	Software integrator; Financial modeller; AI specialist, Investor; Trader; Project manager
Preconditions	• Financial historical data samples, mapping to KG and historic graph creation; • Operational relationships, correlations, and pattern detection; • Missing data interpolations and quality rule implementation; • Synthetic data generation capability; • Inter-changeable synthetic data capabilities for different financial domains (e.g., financial markets, banking, insurance, SWIFT).
Success conditions	• Computational efficiency and robustness for producing synthetic data; • Similar synthetic data quality and structure to real historical data; • Similar financial algorithm behaviour when using both synthetic and historical data.
Failure conditions	• Low-quality synthetic data, not meeting the quality requirements.

Documentation

Functionality	Quality financial rules implementation
Requirement identifier	UC-1.5
Requirement name	Quality financial rules implementation
Priority	Very high
Description	Implement independent quality rules and the identified correlations and patterns (also as runes) for generating synthetic data.
Precondition	Graph-Optimizer; UC-1.4
Source	UC-1 description
Category	Non-functional; Domain; Quality.

4.2.2 UC-2: Global Foresight for Environmental Protection

Description and mapping

Table 21: UC-2 details: global foresight for environmental protection.

Identifier	UC-2
Title	Global foresight for environmental protection
Source	ER; State-of-the-art analysis; Focus group; Stakeholder interviews; User stories.
Tags	News analytics, real-time event screening, ESG analytics, KG.
Description	UC-2 harnesses the power of graph technology to analyse vast amounts of data from various open web sources, create an intuitive and interactive contextual KG that empowers decision-makers to stay ahead of emerging trends and scenarios and make informed, data-driven policy decisions that will positively impact the environment.
Responsible	ER
Stakeholders	Risk analysts; AI specialists; Macro-investors; Geopolitical analysts
Preconditions	• Open-web sources data; • GNN (PyG) integration with Graph-Scrutinizer for dataset optimisation; • Graph Greenifier tool supporting GNN optimisation; • Graph-Massivizer platform personalised for UC-2.
Success conditions	• Absorb and visualise real-time global news data into massive KGs; • Make informed predictions on predefined ESG topics; • Detect trends in individual industries, sectors, locations, etc; • Predict likely future events for a given list of recent events for a company.
Failure conditions	• Unclear indications in forecasting trends; • Inability to detect patterns in chains of events and unable to predict future events.

Documentation

Functionality	Global news data visualisation
Requirement identifier	UC-2.3
Requirement name	Global news data visualisation
Priority	Medium
Description	Show the sequence of detected events and most likely future events.
Precondition	Graph-Scrutinizer; Graph-Greenifier; UC-2.2
Source	UC-2 description
Category	Non-functional; Domain; Interface; User.

4.2.3 UC-3: Green AI for Sustainable Automotive Industry

Description and mapping

Table 22: UC-3 functionality mapping to Graph-Massivizer tools.

Functionality	Description	Graph-Massivizer Toolkit
Manufacturing KG creation	• Ingest sensor data and an ontology in the Graph-Massivizer platform and convert it to the KG format.	Graph-Inceptor
KG subgraph selection	• Extract a sub-KG relevant to the next prediction, including, for example, the most recent welding spots.	Graph-Scrutinizer
Time series integration	• Transform time series data into a graph structure to enable joint KG analysis for graph prediction.	Graph-Scrutinizer
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

Documentation

Functionality	Manufacturing KG subgraph selection
Requirement identifier	UC-3.7
Requirement name	Manufacturing KG subgraph selection
Priority	High
Description	Employ multiple sample KG strategies to ensure that the remaining subgraph is helpful for predicting the quality of the next welding spot.
Precondition	Graph-Scrutinizer; UC-3.2
Source	UC-3 description
Category	Functional; Domain.

Functionality	Manufacturing time series integration
Requirement identifier	UC-3.8
Requirement name	Manufacturing time series conversion to graph
Priority	High
Description	Convert time series data (i.e. sensor readings) into graph data that encapsulates the temporal dimension.
Precondition	Graph-Scrutinizer; UC-3.3.1
Source	UC-3 description
Category	Functional; Domain.

4.2.4 UC-4: Data Centre Digital Twin for Sustainable Exascale Computing

Documentation

Functionality	HPC monitoring data ingestion
Requirement identifier	UC-4.1.4
Requirement name	Generation of virtual KG for complex graph queries
Priority	Very high
Description	Dynamically generate virtual KG based on query requirements, with only the relevant data points constructed to avoid database persistence, conserving limited disk storage space (which increased by 700 compared to conventional NoSQL approaches upon RDF conversion).
Precondition	Graph-Scrutinizer; UC-3.2
Source	UC-3 description
Category	Functional; Domain.

5 GRAPH-MASSIVIZER ARCHITECTURE DESIGN

5.1 INTRODUCTION

Figure 43 depicts a high-level conceptual overview of the Graph-Massivizer architecture consisting of three main building blocks:

1. **metaphactory platform** provides the entry point to the users of the Graph-Massivizer toolkit through ontology models and interfaces;
2. **Graph-Massivizer toolkit** consisting of two layers:
   - **Graph operational layer** provides BGOs for graph creation, enrichment, query, analytics, and algorithms, composed into a workflow that represents a logical plan describing the abstract execution steps.
   - **Graph processing layer** transforms the logical plan into a physical plan with specific descriptions of the BGO execution steps, such as the implementation, performance, and energy cost estimates, providing optimised, sustainable, energy-aware, serverless graph processing on the heterogeneous infrastructure.
3. **Heterogeneous infrastructure** comprises HPC, Cloud, and Edge devices for scheduling, deployment, execution, and monitoring.

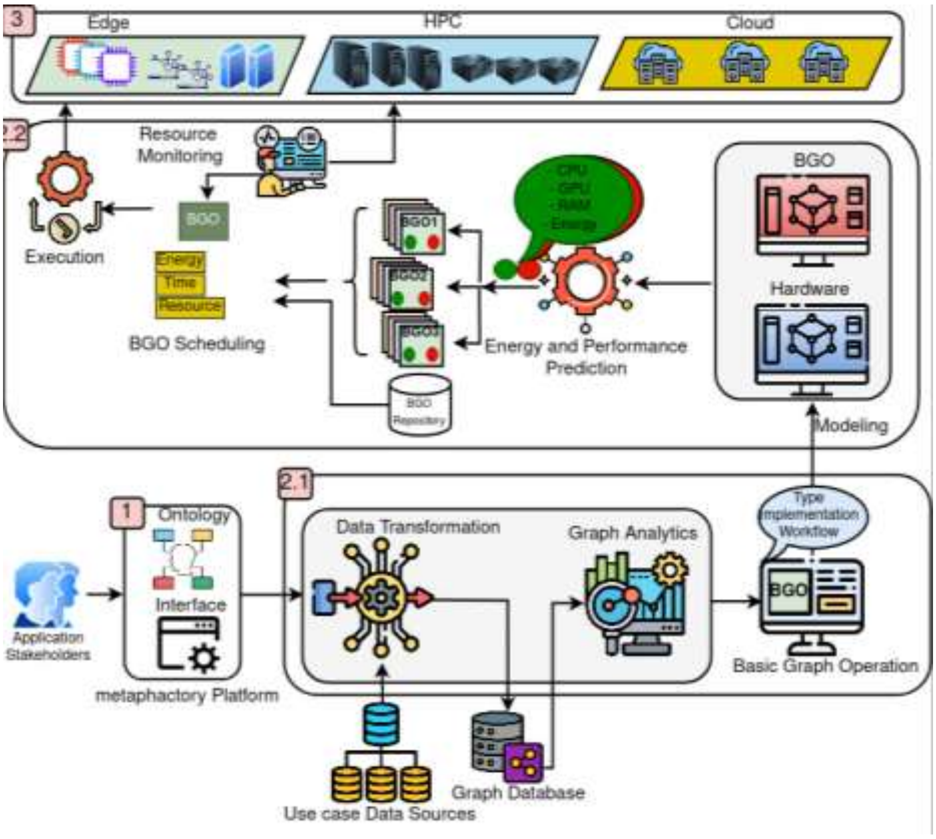

Figure 43: Conceptual high-level overview of the Graph-Massivizer architecture.

The original Graph-Massivizer design described in Deliverable D2.1 proposed a technical C4 architectural model with five tools at different abstraction levels. Figure 44 shows the C4 context diagram[14] that illustrates the tools' high-level functionality and interactions with the metaphactory platform, shared data sources, graph database, computing infrastructure, and stakeholders:

[14] https://c4model.com/

1. **Graph-Inceptor** provides an ETL pipeline supported by BGOs for data ingestion, graph creation, and access;
2. **Graph-Scrutinizer** allows graph analytics comprising heterogeneous BGOs, such as queries, analysis, and ML;
3. **Graph-Optimizer** provides performance and energy consumption estimates of individual BGOs and the full workflow on a single-node system;
4. **Graph-Greenifier** estimates graph processing performance, energy, and sustainability metrics at large data centre levels;
5. **Graph-Choreographer** ensures the correct and successful execution of the graph ingestion and processing tasks.

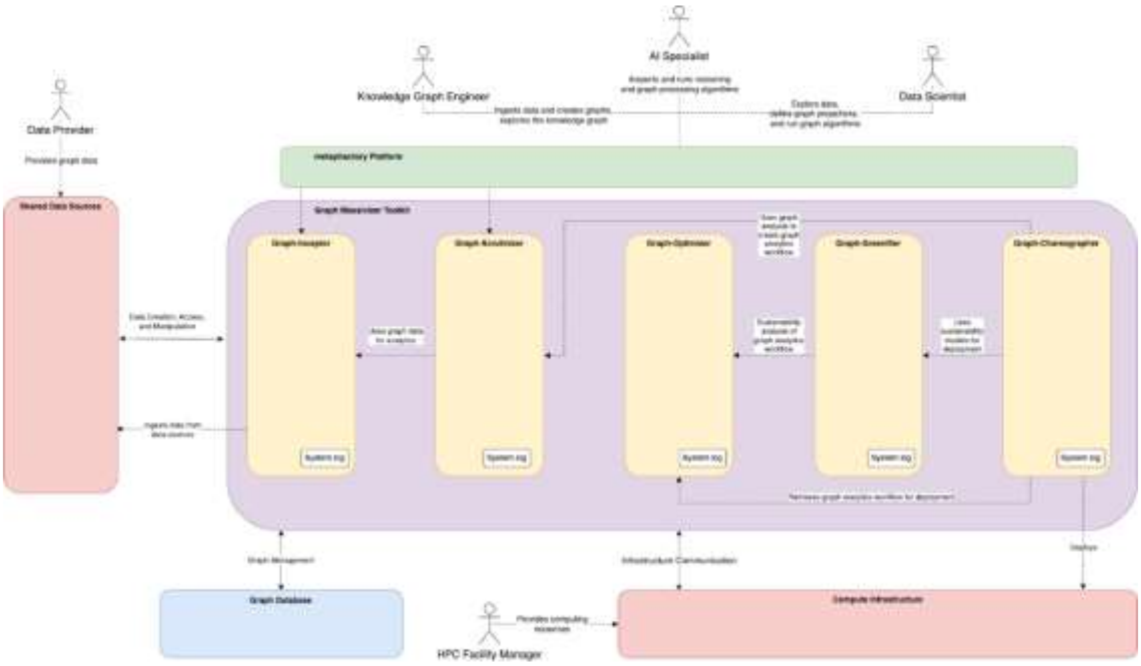

Figure 44: C4 context diagram of the Graph-Massivizer toolkit architecture.

Deliverable D3.1 revised and documented the Graph-Inceptor and Graph-Scrutinizer tools, deliverable D4.1 introduced the Graph-Optimizer, while deliverables D5.2 and D5.3 will cover Graph-Greenifier and Graph-Choreographer.

5.2 TOOLKIT INTERACTION SEQUENCE DIAGRAMS

This deliverable extends the Graph-Massivizer architecture with UML sequence diagrams describing the formal external interaction of the toolkit with stakeholders, the metaphactory platform, data sources, and the infrastructure as it executes.

Figure 3 illustrates the interaction sequence diagrams of the Graph-Massivizer toolkit, further broken down into its five constituent tools in Figure 4, consisting of four steps, applied two main types of stakeholders:

- **Application stakeholders** are the financial, strategic foresight, automotive industry and HPC actors, introduced in the deliverable D2.1;
- **HPC production manager** is a technical o individual (e.g., at CINECA) who deploys and manages the BGOs and Graph-Massivizer tools.

Table 1 summarises the interaction steps using a reference UML type for structure and modularity reasons, further expanded in the following sections.

Figure 45: Graph-Massivizer toolkit external interaction sequence diagram.

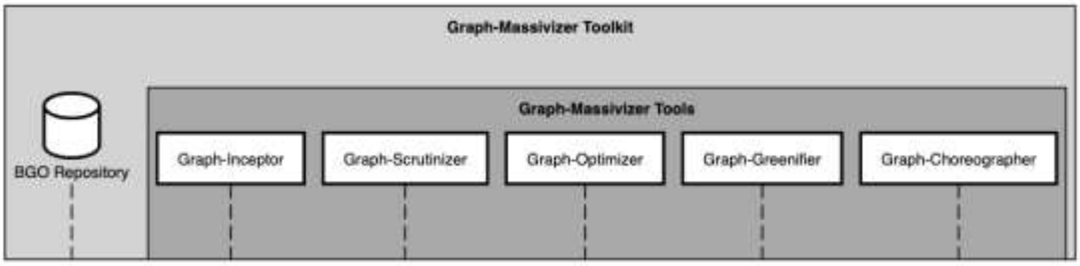

Figure 46: BGO repository and the five Graph-Massivizer tools.

Table 23: Graph-Massivizer toolkit external interaction steps.

Step	Message	Type	Stakeholder Role	Description
1	Register BGO	Reference	HPC production manager	Section 5.3
2	Register data source	Reference	HPC production manager	Section 5.4
3	Define BGO workflow	Reference	Financial, strategic foresight, automotive industry, HPC	Section 5.5
4	Execute BGO workflow	Reference		Section 5.6

5.3 BGO REPOSITORY

The architecture introduces a **BGO repository** to the Graph-Massivizer toolkit containing descriptions of all BGOs and references to available implementations.

5.3.1 BGO Model

The Graph-Massivizer toolkit leverages the **BGO** abstraction as a graph processing unit $BGO(i, f, h, e)$ parameterised to:

- Consider a particular input i;
- Apply a certain operation (or function) f;
- Use certain hardware h and software environments (e.g., programming language) e;
- Produce certain output, not explicitly specified, its identifier and eventual persistence location managed by convention.

Multiple BGOs can form an execution graph, abstracted into a **workflow** comprising **tasks** and enabling complex executions. BGO definitions have **types** to ensure their compatibility when assembled in a workflow.

5.3.2 BGO Implementation

The Graph-Inceptor and Graph-Scrutinizer tools provide most BGO implementations on the Graph-Massivizer platform. Deliverable D3.1 lists identified BGOs required by the use cases and additional BGOs supported by these tools. Graph-Optimizer provides cost estimates at a BGO level on specific hardware, while Graph-Greenifier contextualises the cost of the workflow execution at an entire data centre level.

In addition to the BGO implementations available through the Graph-Massivizer platform, users may similarly register custom BGOs using the metaphactory, described in Section 5.3.4. Upon registering all custom BGOs with metadata about the implemented algorithms and resource locations in the BGO repository, an end user can select amongst all available BGOs interacting with the Graph-Massivizer platform.

5.3.3 BGO Ontology

An ontology represents the semantics of BGOs featured in the BGO repository of the Graph-Massivizer platform. By defining the structure and intended meaning of the relevant parts of BGOs, this ontology allows the toolkit to handle and query metadata regarding potentially available BGOs. The consortium actively develops this ontology in alignment with the toolkit and adjusts it when identifying new requirements.

Figure 47: BGO ontology diagram.

Figure 47 shows the **BGO ontology** diagram comprising its classes and their relations. The class in the centre is `BGO`, which has multiple subtypes with different properties, such as `Creation` BGO for loading data or `Summarization` BGO for producing a simplified graph version. At least one algorithm must implement each `BGO`, expressed by the one-to-many `Algorithm Implementation` relations. A `Data Property` class defines the data input or output from an algorithm, allowing the identification of possible connections between `BGOs` that may occur in sequence. The `Task` and `Workflow` classes express user-defined graph processing applications, where each `Task` corresponds to a `BGO`. Finally, the `Use Case` class relates to other ontologies and connects with data relevant outside the `BGO` model.

5.3.4 BGO Registration

All BGOs part of the Graph-Massivizer toolkit, including custom BGOs, must first register relevant descriptive and implementation information to the metaphactory using either a REST API or a Webform. The metaphactory inserts the BGO registration data into the BGO repository as a specific named graph in the graph database. Figure 48 illustrates the BGO registration sequence diagram consisting of two steps presented in Table 24.

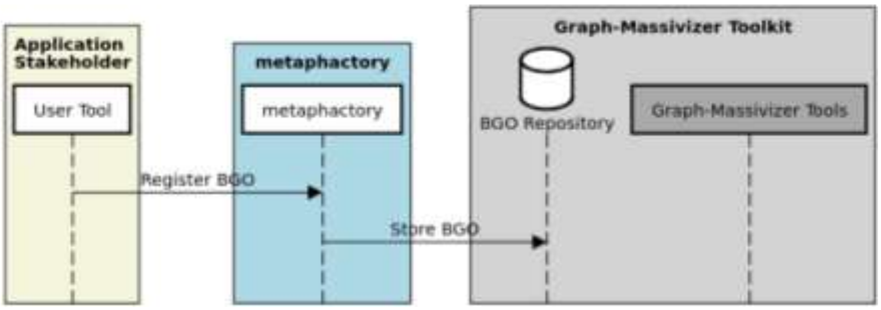

Figure 48: BGO registration sequence diagram.

Table 24: BGO registration steps.

Step	Message	Type	Description
1	Register BGO	REST API / Webform	User registers BGO information containing the name, type, characteristics, and at least one implementation reference compliant with the BGO model and ontology.
2	Store BGO	SPARQL API (insert)	metaphactory stores BGO information in the BGO repository as RDF graph data compliant with the BGO model and ontology.

5.3.5 BGO Lookup

Figure 49 depicts the interactions between Graph-Massivizer tools and the metaphactory platform to search for BGO implementations using the steps detailed in Table 25.

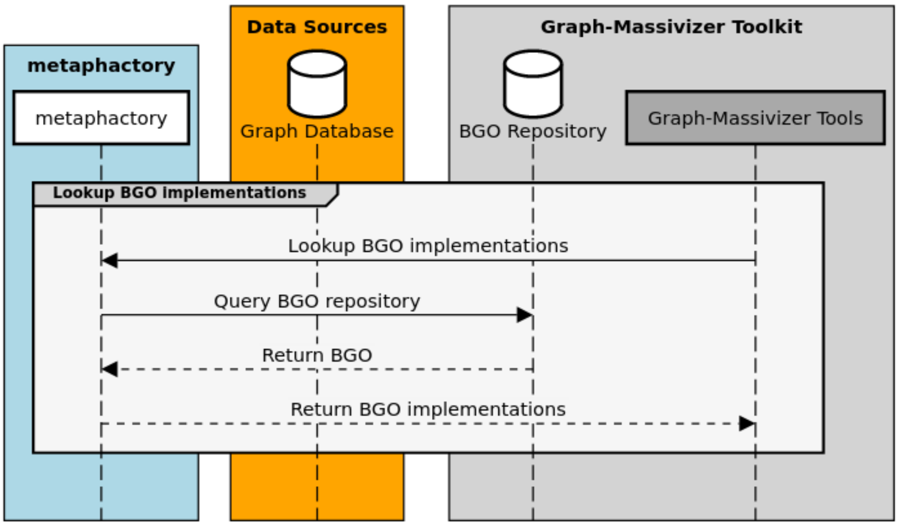

Figure 49: BGO lookup sequence diagram.

Table 25: BGO implementation lookup steps.

Step	Message	Type	Description
1	Lookup BGO implementations	REST API	Graph-Massivizer tools request BGO implementations to metaphactory through an API call, which queries the BGO repository using the following parameters: • Request UUID; • BGO name; • BGO implementations list.
2	Query BGO repository	REST API	
3	Return BGO	Response	
4	Return BGO implementations	Response	

5.4 GRAPH DATABASE

All relevant graph data sources processed by the Graph-Massivizer toolkit must have a **graph database** registration. Deliverable D3.1 describes the graph database as a **virtual KG**, a comprehensive composition of various graph types capable of accessing a persistence layer to retrieve subgraphs or use case data in raw format (e.g., time-series data). Additionally, it can retrieve a sampled or summarised version of the graph to compute approximate or competitive results at a fraction of the cost of processing the whole graph.

5.4.1 Data Registration and Ingestion

Figure 50 represents data source registration through metaphactory and ingestion, automatically performed by the ETL pipeline and GraphMa components of Graph-Inceptor. Table 26 illustrates the details of these processes.

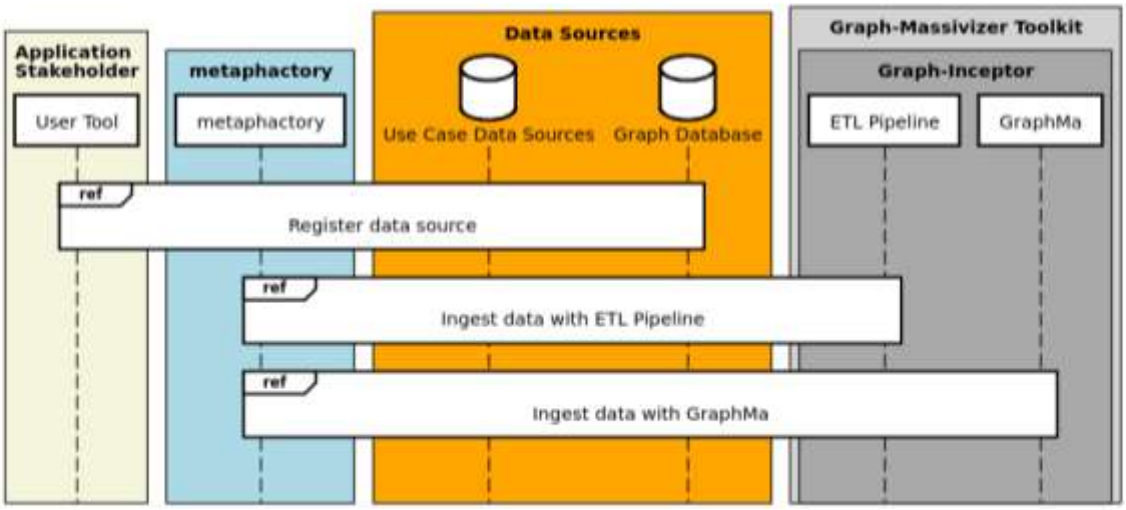

Figure 50: Data registration and ingestion sequence diagram.

Table 26: Data registration and ingestion steps.

Step	Message	Type	Description
1	Register data source	Reference	The user registers data sources.
2a	Ingest data with ETL pipeline	Reference	ETL pipeline and GraphMa components of Graph-Inceptor ingest data.
2b	Ingest data with GraphMa	Reference	

5.4.2 Data Source Registration

Figure 51 shows the sequence diagram for the data source registration consisting of two self-explanatory steps summarised in Table 27.

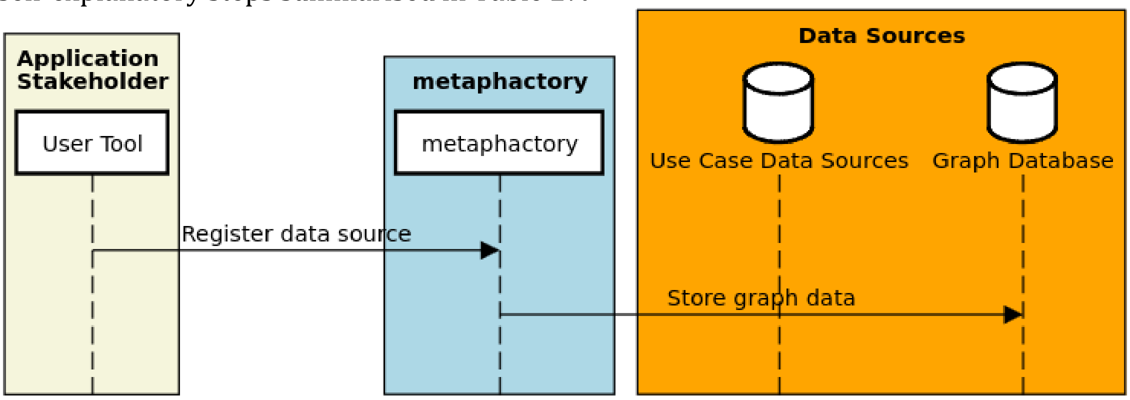

Figure 51: Data source registration sequence diagram.

Table 27: Data source registration steps.

Step	Message	Type	Description
1	Register data source	REST API or Webform	The user registers data source information compliant with the BGO ontology.
2	Store graph data	SPARQL API (insert)	metaphactory stores data source information in the graph database as RDF graph data compliant with the BGO ontology.

5.4.3 ETL Pipeline Data Ingestion

Figure 52 shows the ETL pipeline ingestion sequence steps, further detailed in Table 28. The result of the ingestion is a virtual KG supporting different degrees of ingestion completeness, as described in Deliverable D3.1. The virtual KG may contain all relevant use case data in an RDF graph or only partial data referencing external use case data for later reading or ingestion as part of a BGO workflow execution.

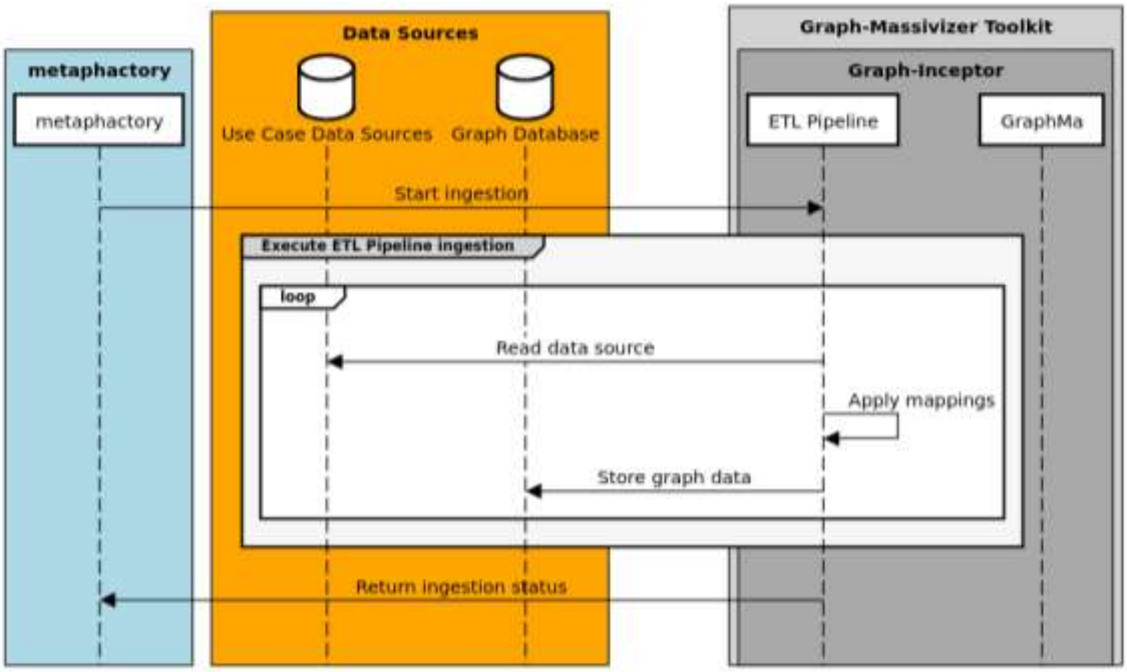

Figure 52: ETL pipeline data ingestion sequence diagram.

Table 28: ETL pipeline data ingestion steps.

Step	Message	Type	Description
1	Start ingestion	REST API / Event	metaphactory starts the ETL pipeline data ingestion.
2	Execute ETL pipeline (read data source, apply mappings, store graph data)	REST API / File transfer	ETL pipeline executes the ingestion process: • Read use case data sources (e.g., files); • Map data onto RDF format; • Insert RDF data into the Graph Database.
3	Return ingestion status	REST API / Event	ETL pipeline returns the status information about the ETL process (e.g., completion).

5.4.4 GraphMa Data Ingestion

Figure 53 shows a typical set of ingestion sequence steps using GraphMa, further detailed in Table 29. GraphMa provides three types of operators, source/traverse, intermediate and terminal, for implementing different BGOs that read, assemble, process, and save graph data in various formats, as described in Deliverable D3.1.

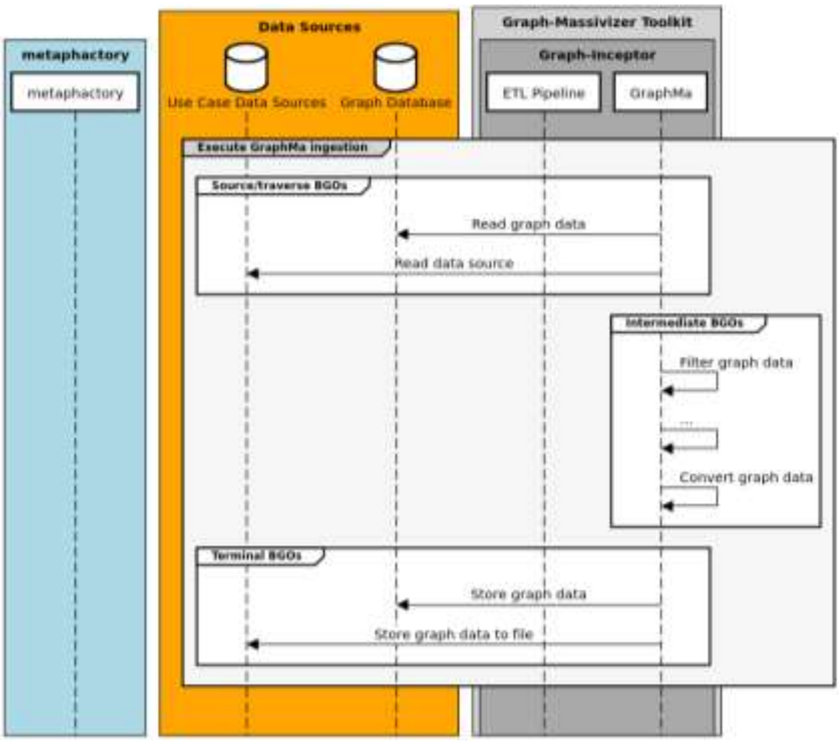

Figure 53: GraphMa data ingestion sequence diagram.

Table 29: GraphMa data ingestion steps.

Step	Message	Type	Description
1a	Read graph data	REST API / File transfer	GraphMa source and traversal BGOs read graph data from the graph database or use case data sources.
1b	Read data source		
2a	Filter graph data	Internal process	Intermediate GraphMa BGOs filter or convert the graph data from one representation format to another.
2b	Convert graph data		
3a	Store graph data	REST API / File transfer	Terminal GraphMa BGOs insert or update graph data in the graph database or write graph data to files.
3b	Store graph data to file		

5.5 BGO WORKFLOW DEFINITION

Figure 54 depicts the sequence diagram for the BGO workflow definition for an input graph, supported through a graphical workflow editor that retrieves the available BGOs from the BGO repository (in the graph database). The application stakeholder submits the workflow to the metaphactory platform for execution, following the steps in Table 30.

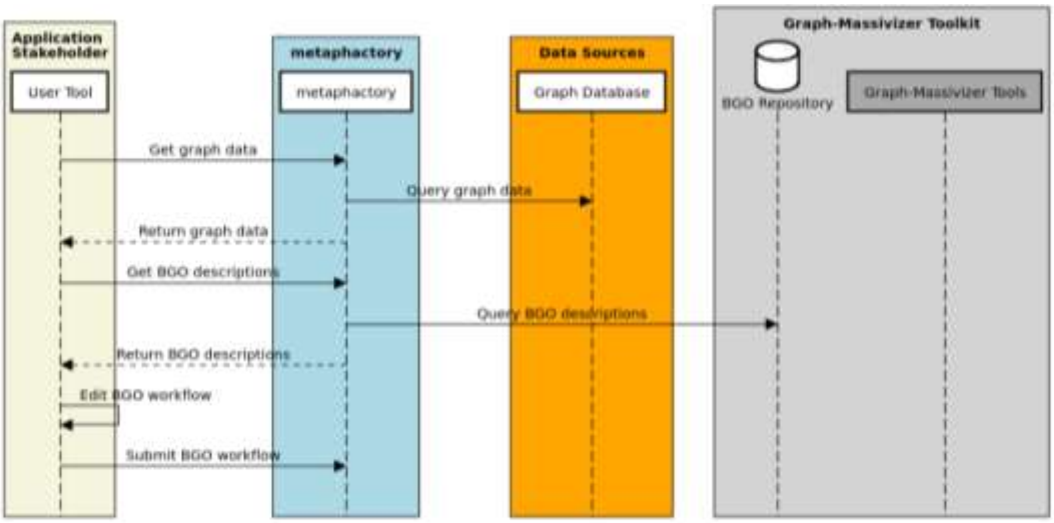

Figure 54: BGO workflow definition sequence diagram.

Table 30: BGO workflow definition steps.

Step	Message	Type	Description
1a	Get graph data	REST API	The application stakeholder calls the metaphactory, which can query the Graph Database to retrieve graph data.
1b	Query graph data		
1c	Return graph data	Response	
2a	Get BGO description	REST API	The application stakeholder calls the metaphactory, which queries a BGO repository to retrieve BGO descriptions.
2b	Query BGO descriptions		
2c	Return BGO descriptions	Response	
3	Edit BGO workflow	Internal process	The application stakeholder edits and completes a BGO workflow.
4	Submit BGO workflow	REST API	The application stakeholder submits the BGO workflow to the metaphactory.

5.6 BGO WORKFLOW EXECUTION

Figure 55 depicts the sequence diagram of the BGO workflow execution submitted by a stakeholder user. The workflow defines a logical orchestration of descriptive BGOs bound to actual implementations, forming a physical execution plan. The BGO workflow goes through several refinement steps for sustainable execution, optimisation, greenifying, scheduling, deployment and executions, summarised in Table 31. Each diagram highlights the homonym message that links it to the previous diagram in the sequence in **blue** and the message that links it to the next diagram in **red**.

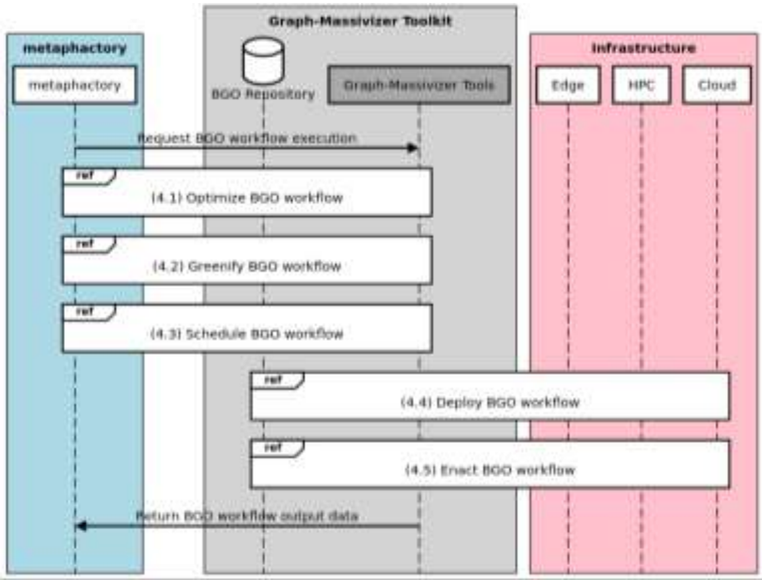

Figure 55: BGO workflow execution sequence diagram.

Table 31: BGO workflow execution steps.

Step	Message	Type	Description
1	Request BGO workflow execution	Event	metaphactory triggers a BGO workflow event containing the following information in JSON format: • Request UUID; • Request hash (to avoid reprocessing); • Input graph identifier for processing; • BGO workflow structure; • BGO identifier list.
2	Optimise BGO workflow	Reference	Graph-Optimizer estimates the BGO execution cost.
3	Greenify BGO workflow	Reference	Graph-Greenifier calculates a green execution plan.
4a	Schedule BGO workflow	Reference	Graph-Choreographer schedules, deploys, and executes a BGO workflow.
4b	Deploy BGO workflow	Reference	
4c	Enact BGO workflow	Reference	
5	Return BGO workflow output data	Event	metaphactory receives the BGO workflow execution status and output data

5.6.1 BGO Workflow Optimisation

Figure 56 depicts the interactions of Graph-Optimizer with the Graph-Massivizer platform to optimise a BGO workflow following the steps summarised in Table 32.

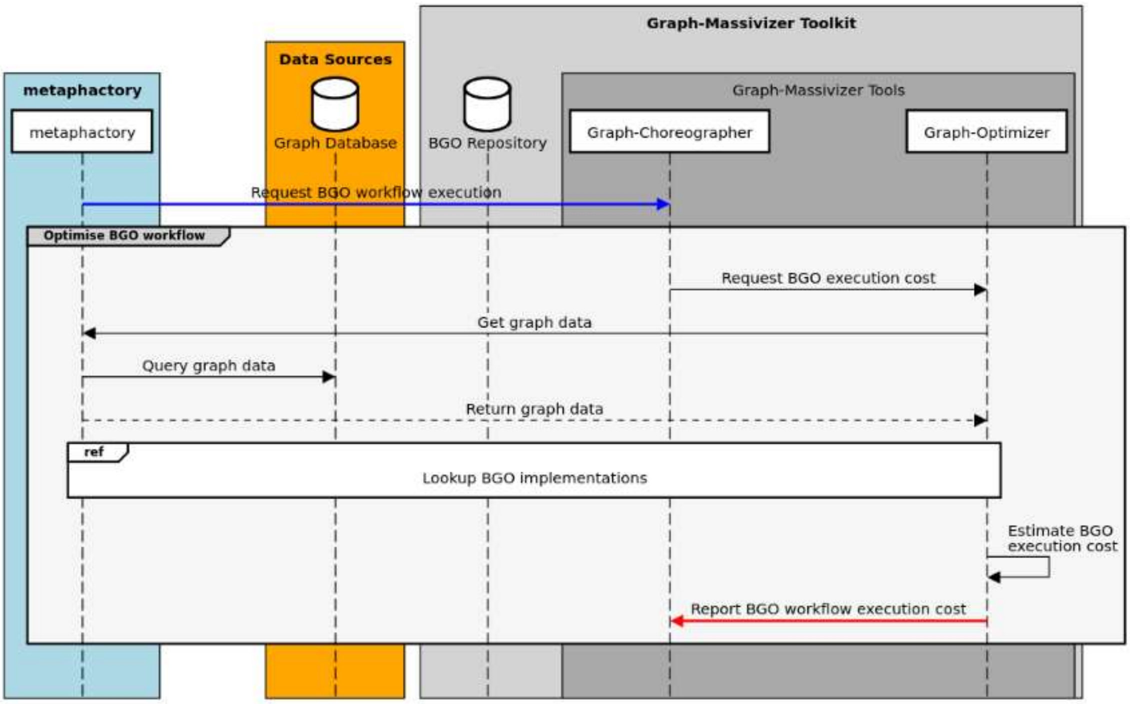

Figure 56: BGO workflow optimisation sequence diagram.

Table 32: BGO workflow optimisation steps.

Step	Message	Type	Description
1	Request BGO workflow execution	Event	Graph-Choreographer receives a BGO workflow execution request from metaphactory (see Figure 55).
2	Request BGO execution metrics	REST API / Event	Graph-Optimizer receives a cost estimate request from Graph-Choreographer containing: • Request UUID; • Input graph identifier; • BGO workflow structure; • BGO identifier list.
3	Get graph data	REST API / Event	Graph-Optimizer retrieves graph data
4	Query graph data	REST API / Event	metaphactory queries graphs from the graph database.
5	Return graph data	Response	metaphactory returns graph data to Graph-Optimizer.
6	Lookup BGO implementations	Reference	Graph-Optimizer retrieves BGO from the BGO repository.
7	Estimate BGO execution metrics	Internal process	Graph-Optimizer calculates or looks up precalculated BGO cost estimates in the BGO repository.
8	Report BGO workflow execution metrics	Reference	Graph-Optimizer sends the BGO workflow cost estimates to Graph-Choreographer containing: • Request UUID; • BGO identifier list and their execution estimates.

5.6.2 BGO Workflow Greenifying

Figure 57 depicts interactions of the Graph-Greenifier with the Graph-Massivizer platform to propose alternative BGO workflow plans following the steps summarised in Table 33.

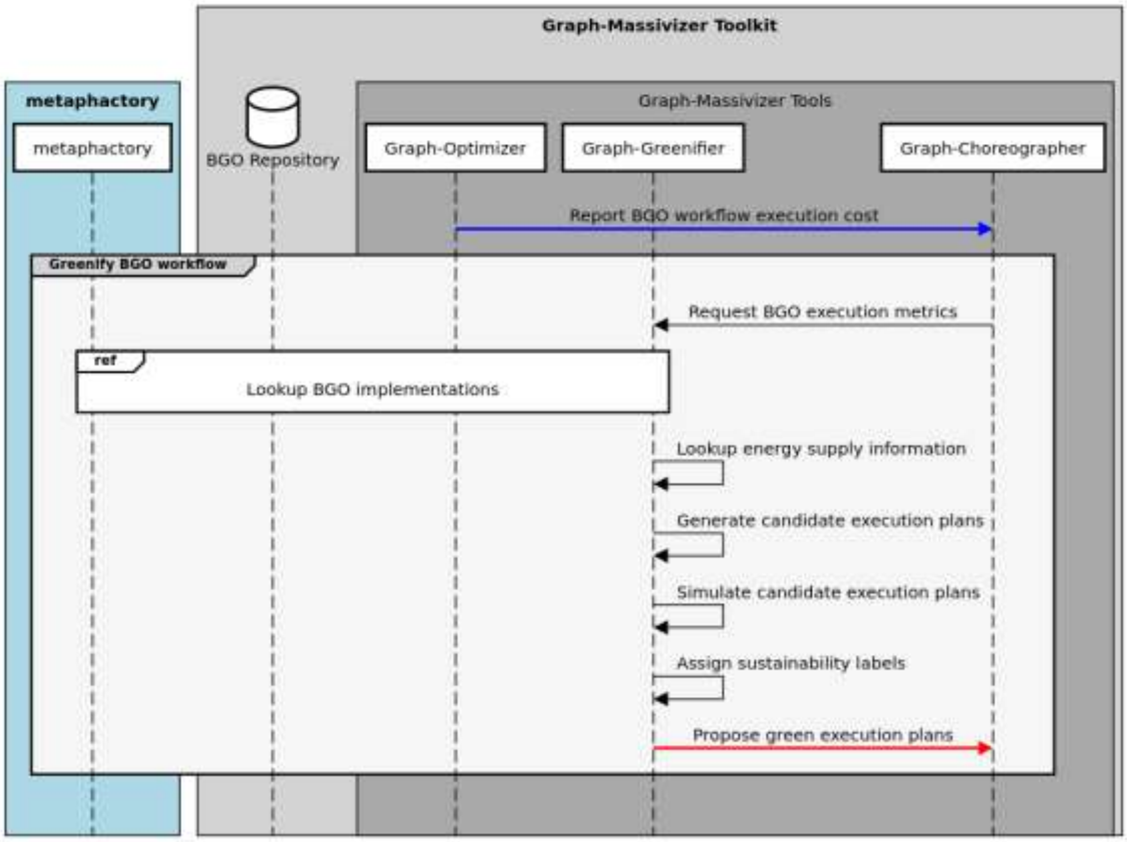

Figure 57: BGO workflow greenifying sequence diagram.

Table 33: BGO workflow greenifying steps.

Step	Message	Type	Description
1	Report BGO workflow execution metrics	REST API / Event	Graph-Choreographer receives estimated BGO workflow execution metrics from Graph-Optimizer (see Figure 56).
2	Request BGO execution metrics	REST API / Event	Graph-Greenifier receives a green execution plan request from Graph-Choreographer containing: • Request UUID; • Input graph identifier; • BGO workflow structure; • BGO identifier list and their estimated execution metrics; • Hardware device list with type (e.g., CPU, GPU), version (e.g., Nvidia A100), architecture (e.g., Ampere, x86), etc.
3	Lookup BGO implementations	Reference	Graph-Greenifier looks up for available BGO implementations (Figure 49, Section 5.3.5).
4	Lookup energy supply information	REST API	Graph-Greenifier looks up historical energy supply information and forecasts in public sources (e.g., ENTSO-E).
5	Generate candidate execution plans	Internal process	Graph-Greenifier generates alternative green execution plans considering BGO execution costs and available hardware.
6	Simulate candidate execution plans	Internal process	Graph-Greenifier simulates the candidate execution plans and quantifies their sustainability and performance.
7	Assign sustainability labels	Internal process	Graph-Greenifier assigns sustainability labels to candidate plans based on simulation and stakeholder requirements.
8	Propose green execution plans	REST API / Event	The Graph-Greenifier publishes the alternative BGO workflow execution plans containing: • Request UUID; • List of execution plans, mapping each BGO implementation to a hardware device with sustainability labels.

5.6.3 BGO Workflow Scheduling

Figure 58 depicts the interactions of the Graph-Choreographer with the Graph-Massivizer platform to schedule a BGO workflow plan based on the propositions received from the Graph-Greenifier following the steps summarised in Table 33.

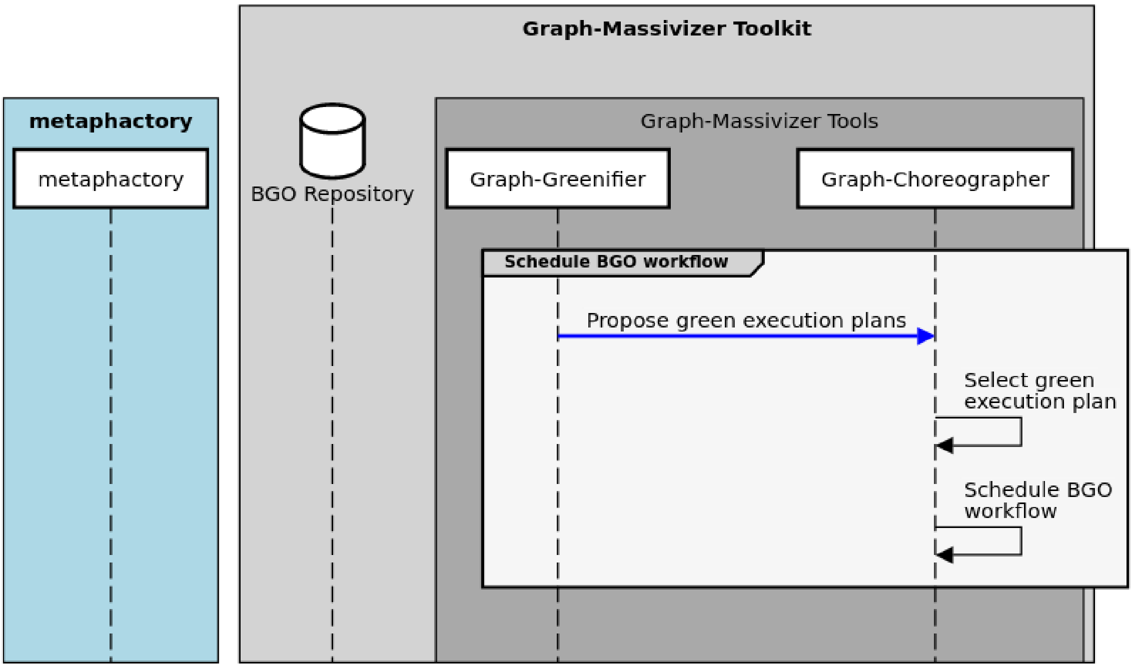

Figure 58: BGO workflow scheduling sequence diagram.

Table 34: BGO workflow scheduling steps.

Step	Message	Type	Description
1	Propose green execution plans	REST API / Event	Graph-Choreographer receives a list of green BGO workflow execution plans from Graph-Greenifier (see Figure 57).
2	Select green execution plan	Internal process	Graph-Choreographer selects the most appropriate green execution plan or recalculates one to ensure sustainable workload orchestration considering resource availability, energy efficiency, execution constraints and objectives.
3	Schedule BGO workflow	Internal process	Graph-Choreographer schedules the BGOs from the selected execution plan to available hardware devices based on the following criteria to ensure a balanced workload distribution: • Resource availability; • Hardware compatibility; • Execution priorities; • Performance and energy constraints.

5.6.4 BGO Workflow Deployment

Figure 59 shows the internal loop executed by the Graph-Choreographer to deploy the BGO workflow on the hardware devices indicated by the schedule, using the steps in Table 35.

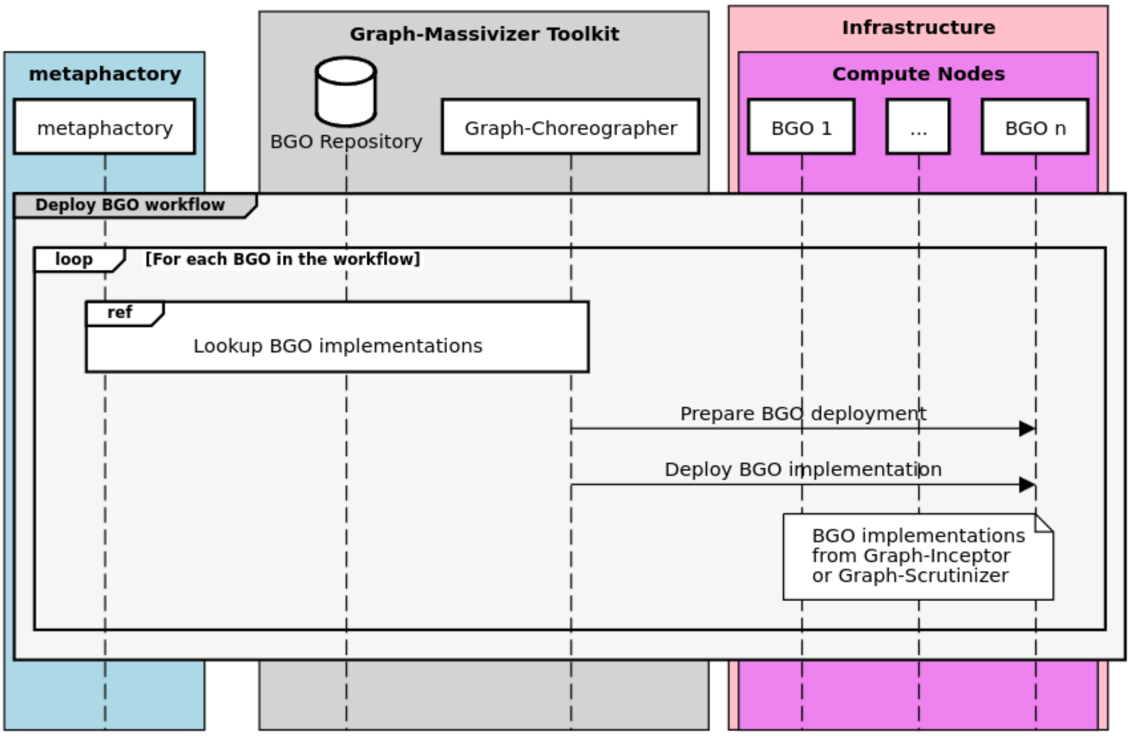

Figure 59: BGO workflow deployment sequence diagram.

Table 35: BGO workflow deployment steps.

Step	Message	Type	Description
1	Lookup BGO implementations	Reference	Graph-Choreographer looks up and downloads the BGO implementations from the BGO repository (Figure 49).
2	Prepare BGO deployment	Internal process	Graph-Choreographer prepares the BGO workflow for deployment, ensuring proper configuration and compatibility with the scheduled infrastructure, including: • Resolving and installing required software dependencies; • Configuring runtime environments, such as containers, virtual machines, or native binaries; • Validating BGO package compatibility with the target hardware architecture (e.g., x86, ARM); • Packaging BGO implementations in a deployment format.
3	Deploy BGO implementation	Internal process	Graph-Choreographer deploys the prepared BGO implementation to the selected hardware devices, including: • Transferring BGO packages or containers to assigned hardware; • Initialising the execution environments (e.g., containers, virtual machines, or native runtime environments); • Validating resource availability for execution, including memory, storage, and computation; • Verifying deployment correctness by pre-execution checks.

5.6.5 BGO Workflow Execution

Figure 60 shows the internal steps executed by the Graph-Choreographer to deploy the workflow BGOs on the hardware devices indicated by the schedule, detailed in Table 36.

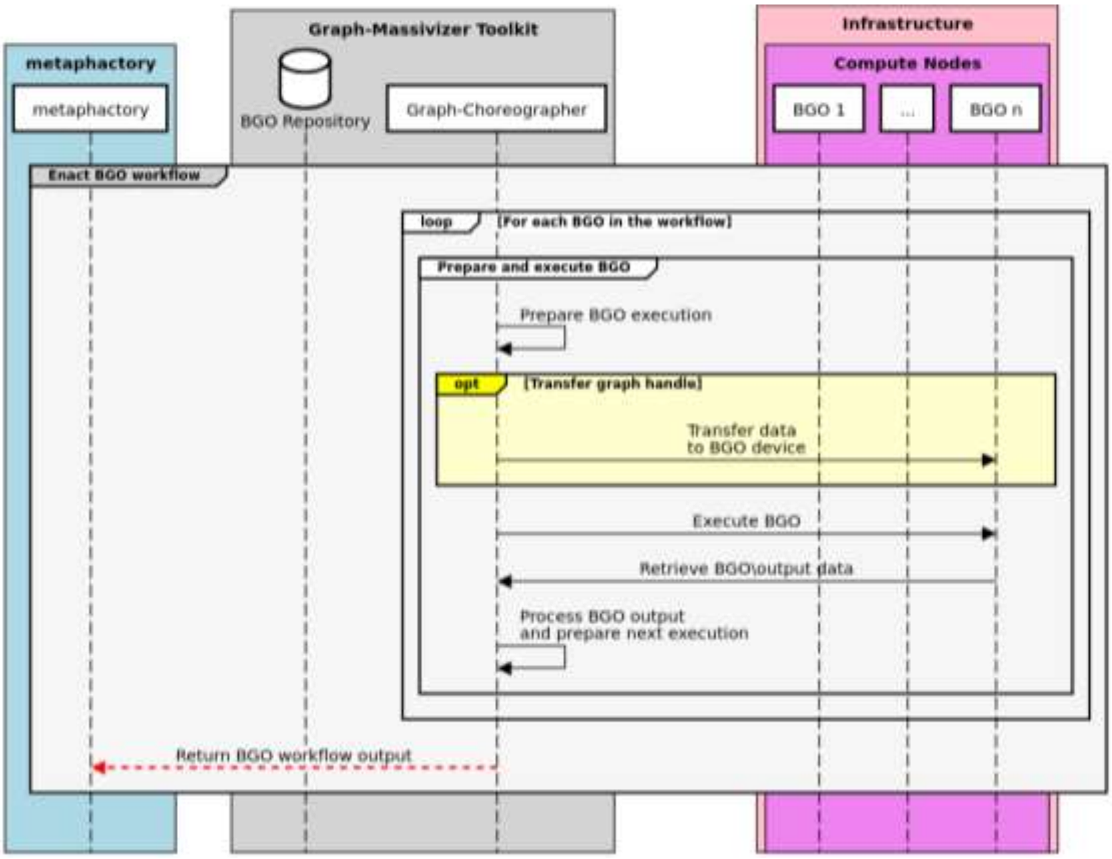

Figure 60: BGO workflow execution sequence diagram.

Table 36: BGO workflow execution steps.

Step	Message	Type	Description
1	Prepare BGO execution	Internal process	Graph-Choreographer prepares for executing the BGO on the hardware computing device, including: • Initialising the BGO workflow runtime settings; • Allocating computing, memory, and I/O resources; • Ensuring compatibility with the BGO workflow's execution requirements defined in the deployment plan.
2	Transfer data to BGO device	File transfer	Graph-Choreographer optionally transfers the BGO input data to the local storage of the executing computing device.
3	Execute BGO	Event	Graph-Choreographer executes the BGO implementation on its deployment device, involving: • Loading the preconfigured runtime environment; • Starting the BGO executions in the workflow according to the defined sequence; • Monitoring BGO execution progress and detecting errors; • Handling runtime errors or interruptions.
4	Retrieve BGO output	Reference	Graph-Choreographer executed the deployed BGO workflow.
5	Process BGO output and prepare the next BGO execution	Internal process	Graph-Choreographer processes the BGO output data, validates it, and prepares the inputs for the next BGO in the workflow.
6	Return BGO workflow output	Event	Graph-Choreographer sends the BGO workflow output data to the metaphactory platform, including: • Packaging the output data in the required transfer format; • Transmitting the output data to the metaphactory platform; • Confirming the successful data transfer.

6 ACADEMIC KNOWLEDGE BGO WORKFLOW EXAMPLE

This section provides an example use case with sequence diagrams that align with the Graph-Massivizer architecture. We use an **academic KG BGO workflow**, referred to as use case 0 (UC-0), initially introduced in deliverable D6.1 to illustrate the four major execution steps using the Graph-Massivizer toolkit execution, serving as an example for the four industrial use cases prototypes planned at month 27. The academic knowledge workflow operates on an open-access KG database of scholarly publications and author information called **SemOpenAlex**[15], attempting to find connections between influential researchers in the same field by tracing co-authorship relations.

6.1 BGO WORKFLOW DEFINITION

Figure 61 shows an example of an academic knowledge BGO workflow specified in three layered phases in the metaphactory platform by an academic reviewer stakeholder.

BGO workflow definition is a sequence of six tasks mapped to the corresponding BGOs and their implementations with the associated data. It finds the central researchers in a given field with the paths between them in sequential execution. More complex workflows in the directed acyclic graph patterns are generally possible.

BGO definition represents the individual processing tasks specified by the stakeholder as part of the workflow engineering process. By identifying BGOs in the workflow definition, the Graph-Massivizer toolkit can leverage all valuable metadata provided by the HPC production manager during the system configuration.

BGO implementation is part of algorithmic graph processing software packages and libraries with different types of input and output and hardware requirements along with the related associated metadata. Thus, an application stakeholder can input a workflow by constructing the sequence of tasks and selecting BGOs, and the toolkit can select the BGOs from the available implementations for execution.

Figure 61: Academic knowledge BGO workflow.

6.2 BGO WORKFLOW EXECUTION

Figure 62 provides the sequence diagram with the high-level execution steps of the academic knowledge BGO workflow, detailed in Table 15, including the concrete contained BGOs as the main distinguishing characteristics among the other use cases.

[15] https://semopenalex.org/

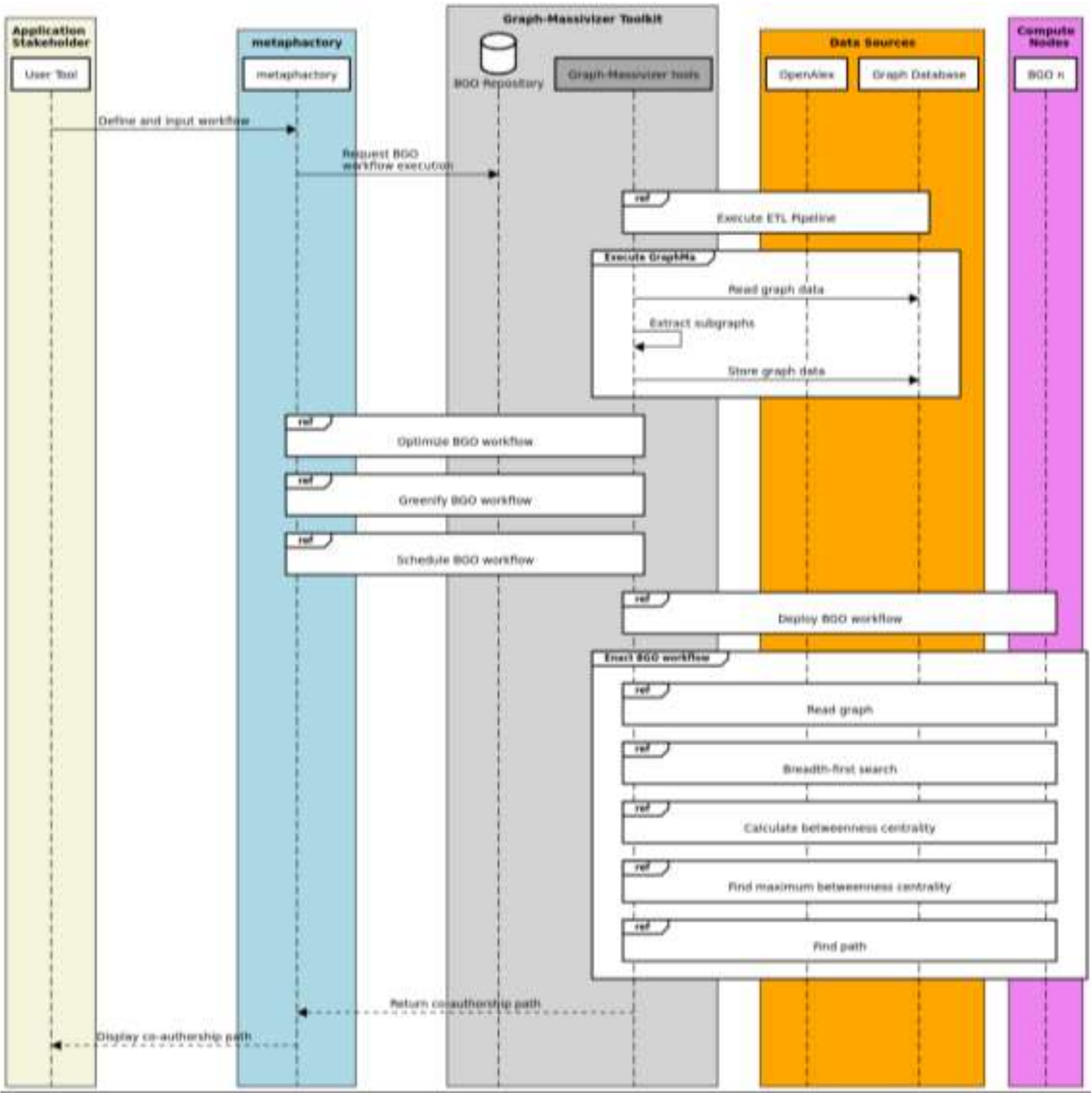

Figure 62: Academic knowledge BGO workflow sequence diagram.

Table 37: Academic knowledge BGO workflow execution steps.

Step	Message	Type	Description	Contextual Step
1	Define and input workflow	REST API / Webform	The application stakeholder inputs a BGO workflow with the metaphactory platform, which identifies the BGO implementations in the repository.	BGO workflow definition
2	Request BGO workflow execution	Event	metaphactory receives input workflow definition, constructs its RDF graph, and sends it to the Graph-Massivizer toolkit.	BGO workflow execution
3	Execute ETL pipeline	Reference	ETL pipeline reads the OpenAlex structured catalogue, applies RDF mappings, and writes them to a SemOpenAlex graph database.	ETL pipeline
4	Execute GraphMa	Reference	GraphMa extracts smaller subgraphs from the SemOpenAlex RDF graph for different inputs.	GraphMa
4.1	Read graph data	Reference	GraphMa reads the SemOpenAlex RDF graph from the graph database.	
4.2	Extract subgraph	Internal process	GraphMa extracts a subgraph based on the user input.	
4.3	Store graph data	Internal process	GraphMa inserts the subgraph into the graph database for later reuse.	
5	Optimise BGO workflow	Reference	Graph-Optimizer estimates BGO execution metrics.	Graph-Optimizer
6	Greenify BGO workflow	Reference	Graph-Greenifier calculates a green workflow execution plan.	Graph-Greenifier
7	Schedule BGO workflow	Reference	Graph-Choreographer schedules the BGO workflow.	Graph-Choreographer
8	Deploy BGO workflow	Reference	Graph-Choreographer deploys the BGO workflow on computing devices.	Graph-Choreographer, metaphactory
9	Enact BGO workflow	Reference	Graph-Choreographer loads sub-graphs from the graph database and processes them according to BGOs.	Graph-Choreographer
9.1	Read graph	Reference	The BGO workflow loads a subgraph from the graph database.	BGO workflow execution
9.2	Breadth-first search	Reference	The BGO workflow executes the breadth-first search algorithm for scanning relevant subgraph nodes.	
9.3	Calculate betweenness centrality	Reference	The BGO workflow executes the betweenness centrality algorithm for selecting relevant nodes.	
9.4	Find maximum betweenness centrality	Reference	The BGO workflow determines the node with maximum betweenness centrality.	
9.5	Find path	Reference	The BGO workflow finds a graph path between an input node and one with maximum betweenness centrality.	
10	Return co-authorship path	Event	Graph-Choreographer returns the path to the popular researcher to metaphactory.	
11	Display co-authorship path	Event	metaphactory visualises the output path of the BGO workflow.	

7 CONCLUSIONS

This deliverable performed an intermediate requirements analysis of the Graph-Massivizer platform, tools and UCs following the first beta-testing and validation campaign completed at M15 and revised the architecture design accordingly. The deliverable successfully covered the following tasks, which are essential for the following project activities:

- **Beta-testing campaign description** comprising the methodology, commonalities and final instantiation for each constituent tool of the Graph-Massivizer toolkit;
- **First beta-testing campaign results with real users** giving fundamental feedback to the tools regarding their requirements evolution, architecture revision and subsequent development iterations;
- **Graph-Massivizer platform and toolkit requirements revision**, logically derived from the beta-testing feedback received during the validation campaigns;
- **Use case requirements revision** following recent state-of-the-art and market analysis;
- **Revised architecture design** of the operational Graph-Massivizer platform, focusing on the interactions among the internal tools and with the external metaphactory platform and its stakeholders;
- **Academic knowledge workflow** illustrating the internal tool integration and external tool interactions with the metaphactory platform and stakeholders.

This deliverable is a **live document**, constantly enhanced and improved during the project, with one more **final revision** planned at M36 (D2.3), as follows

- **Continuously monitor and expand the state-of-the-art** as new technologies emerge on the market combined with a corresponding risk analysis and mitigation strategy;
- **Iteratively evolve the requirements** of the UCs and the Graph-Massivizer platform by collecting feedback from the two remaining validation campaigns of the integrated prototypes conducted at M27 and M33;
- **Update and improve the software architecture** of the Graph-Massivizer platform and its constituent tools based on internal and external feedback received;
- **Evaluate the objective KPIs** in the final validation campaign (at M33, using the last platform architecture and prototype release) to assess the project's success.

The partners intend to identify **other innovative applications**, scenarios and implementations derived from the original four UCs to enhance the relevance of the Graph-Massivizer project.

ACRONYMS

Acronym	Description
AI	Artificial Intelligence
API	Application Programming Interface
BGO	Basic Graph Operation
CSV	Comma-Separated Values
D<*number*>	Deliverable <*number*>
ESG	Environment, Social and Governance
ETL	Extract-Transform-Load
GNN	Graph Neural Network
HPC	High-Performance Computing
I/O	Input / Output
JSON	JavaScript Object Notation
KG	Knowledge Graph
M<*number*>	Month <*number*>
ML	Machine Learning
MS	Milestone
RDF	Resource Description Framework
REST	Representational State Transfer
UC	Use Case
UUID	Universally Unique Identifier
WP	Work Package
XML	Extensible Markup Language