VISDOM reference architecture

Deliverable 2.5.2

Version history

Version	Date	Author	Notes
0.1	12.02.2020	Kari Systä (TAU)	Initial draft based on discussions with Mika Koivuluoma, Markus Kelanti, and Henri Bomström.
0.2	24.02.2020	Kari Systä (TAU)	Responses to comments from Outi Sievi-Korte
0.3	09.03.2020	Kari Systä (TAU)	Solved all comments from Outi, added a few lines about security, privacy and access control
0.4	04.05.2020	Kari Systä (TAU)	More about dashboard composition
0.5	10.05.2020	Kari Systä (TAU)	Added more text to intro to explain the purpose and scope.
0.6	12.05.2020	Outi Sievi-Korte (TAU)	Added text on micro-frontends.
0.7	18.05.2020	Kari Systä	Small bug fixes
0.8	19.05.2020	Kari Systä	Notes from the plenary/workshop
0.9	25.05.2020	Kari Systä	Further notes from plenary/workshop
0.91	08.09.2020	Kari Systä	Towards deliverable 2.5.1
0.92	10.09.2020	Henri Bomström	Dashboard architecture description.
0.93	13.09.2020	Vivian Lunnikivi	Several small fixes and comments
0.94	13.09.2020	Kari Systä	Responses to comments
0.95	14.09.2020	Henri Bomström	Multiple fixes based on comments.
0.96	15.09.2020	Henri Bomström	Revisited sections 2.2, 2.4, and 3.
0.97	16.09.2020	Henri Bomström	Revisited section 2 and fixed inconsistencies.
1.00	21.09.2020	Kari Systä	Checked latest changed, moved version to 1.0,
2.01	23.03.2022	Kari Systä	Towards D2.5.2
2.02	28.04.2022	Kari, Henri, Ville, Markus	Meeting in Oulu collaborate with the document.
2.03	02.05.2022	Ville Heikkilä	Updates regarding decriptions for fetcher, adapter, broker, and data links
2.04	04.05.2022	Kari Systä	Some re-organization and started to add demo descriptions
2.05	09.05.2022	Ville Heikkilä	Some cleaning up on the text and comments
2.06	16.05.2022	Ville Heikkilä	Revisions throughout the document.
2.07	27.05.2022	Ville Heikkilä	Revisions for the analysis chapter + conclusions.

Table of Contents

Executive summary

TODO

1 Introduction

This document describes the top-level reference architecture of VISDOM. The purpose of this document is to

- provide a reference architecture that encapsulates the research results, and
- recognize and document exploitable or standardizable components and interfaces,
- It should be noted that this technical reference architecture is assumed not to be implemented as such in each partner specific implementation.

1.1 Business requirements

The following requirements should be supported by the architecture:

- The architecture shall support stakeholder and company specific dashboards that are composed of new visualizations developed in VISDOM project.
- The visualizations are usually based on several data sources (i.e. tools) and hence provide synthesized and abstracted views. The mainstream software engineering tools include visualizations about their data, one the main innovations of VISDOM is to combine data from several tools.
- The architecture shall support zooming-in/investigating" the problems. This means that the users can investigate the original reason behind the interesting detail discovered from the visualization. This is mainly a responsibility of the visualizations but implies that the "zooming in" needs access to all data used in the visualizations.
- The architecture shall support a large and extendible set of data sources (tools). This means that it should be possible to dynamically add new tools as data sources.
- The architecture shall support all activities in DevOps development. This means that stakeholders interested in business, management, development, technology, deployment, hosting and operation should be supported. Furthermore, tools and data sources related to all these aspects can be used.
- The architecture shall support research and development in the VISDOM project. This means that we can test and demonstrate integrations with external and internal components in an agile way.
- The architecture shall support use of industry and de-facto standards. This mostly involves the data fetchers - use of standard data and/or standard data interfaces (such as ODATA) improves re-usability of the fetchers. Some, standards include KPIs (for example DORA DevOps metrics). This KPIs could be calculated by the adapters.
- The architecture shall support exploitation of the results. This means that developed ideas and components can be exploited in several contexts individually or together.

1.2 Architecture as described in the full project proposal

The technology value chain, as specified in the VISDOM Full Project Proposal (FPP), is depicted in the Figure 1:

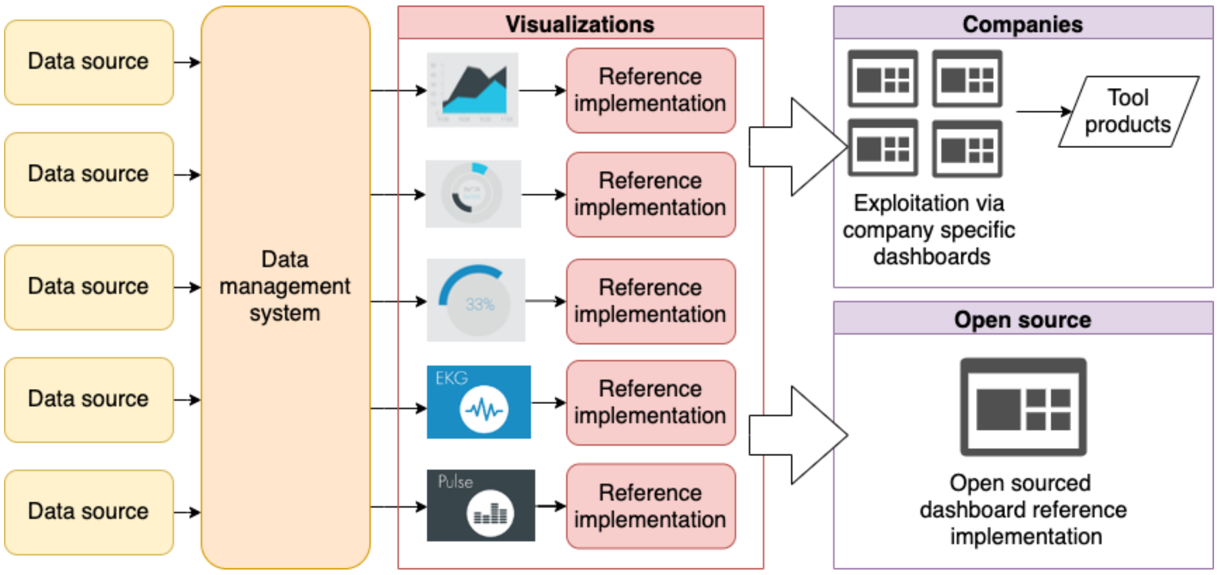


Figure 1 Technology value chain of the VISDOM project

The components in Figure 1, re-drawn from the FPP Figure 4, are the following:

- Data sources are the different development tools, databases and repositories used in software engineering. In VISDOM we develop methods and tools (open source by default) for collecting data from a variety of existing sources. We will also create a framework and instructions for developers who want to integrate new data sources.
  Architecture note: the data sources are external to VISDOM and we are dependent on the interface and data formats supported by the tools.
- The **data management system** includes ways to unify, merge, link and store the input data for effective visualizations and possible further processing. The data models and analysis methods will be realized as running software and included in the reference implementation, making it possible to use the resulting metrics even without visualization.
  Architecture note: the data management system is the most challenging part of the architecture and thus it is the main topic of this document.
- The project produces a set of evaluated and piloted **visualizations with reference implementations**[1]. The designs include implementation guides, guidelines for users and requirements for the data management solution. The designs can be accompanied with reusable reference implementations but can also support independent implementations on the same data.
- The project creates a configurable **dashboard** concept that can display different content and views to different stakeholders. We will also create prototype dashboards for the pilots, and the companies may exploit these dashboards by integrating them in their current and future development support tools. A dashboard is not necessarily just for

[1] https://github.com/visdom-project/visdom

direct presentation of data, but it can also include diagnostic tools and help the stakeholders investigate possible problems in their projects.

2 The VISDOM architecture

This section describes the general architecture specified during the project. The first version was created after the first 12 months of the project, and the final architecture was created after pilot implementation in various use cases. The presented reference architecture results in slightly different real-world implementations for individual cases. Thus, we also specify possible implementation related issues and points of consideration as a part of the solution to highlight a need for customization per each unique case. One of the most important aspects for consideration includes the centralization of different components in the architecture. For example, to avoid having single monolithic components that handle each and every situation and need, the actual implementation may feature multiple customized versions of the components deployed in parallel. As an example, a real-world implementation of the architecture may utilize one or more data management systems to match the specific needs of certain teams while having a single data management system serve other users. As a final note, lessons learned and further research questions are presented at the end of this document.

The overall architecture consists of three main elements: data management, visualizations, and dashboards. First, the data management system is responsible for providing a unified interface (API, data model, etc.) for visualization to access data, and facilitates dynamically adding new tools as data sources. Second, visualizations represent generic components that may offer advanced functionality of their own, such as zooming in to the problem root causes, and can be configured based on stakeholder needs and the underlying dashboard.

Lastly, the dashboard composer is responsible for matching visualizations to user needs with customizable views. The composer decouples visualization logic from the dashboard itself, allowing the architecture to support both stakeholder and company specific dashboards that can support all activities in DevOps development.

2.1 Data architecture

The data management system fetches data from various tools used in software engineering and provides a uniform data interface for visualizations to consume. The following figure describes the general idea of the data architecture.

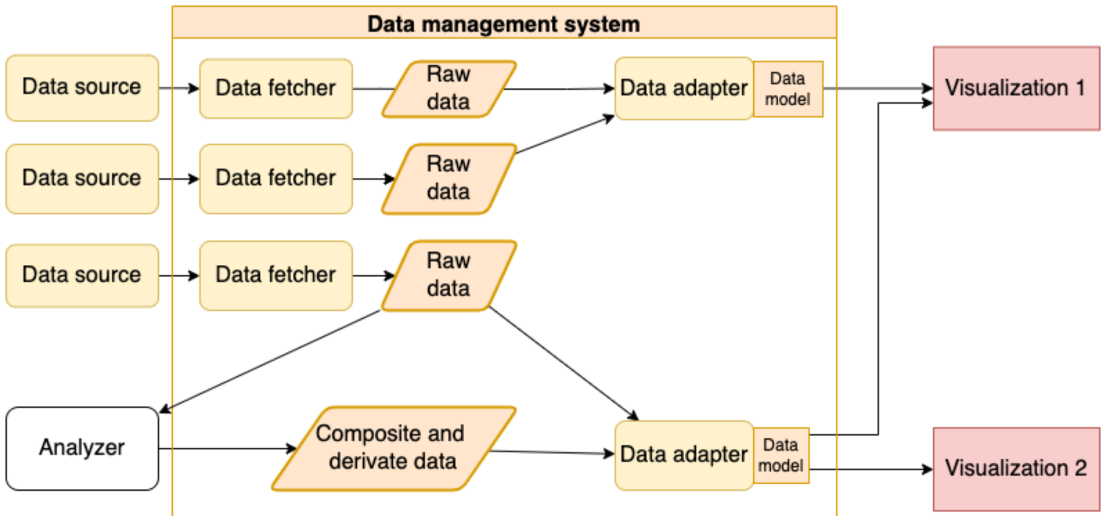


Figure 2 Data architecture

The components of the data architecture are data sources, data fetchers, raw data, and data adapters.

- **Data sources** are typically SW engineering tools that produce data, but our system is flexible for other types of data sources, too. In particular, various analysis tools (**Analysers**) can compose new data items, composites and derivates, based on existing raw data items. Also, a service or a product can generate additional data from the operation by customers or from various tests, and can thus work as a data source. The VISDOM project aims to develop such tools and enable the use of external tools. The architecture shall support both push and pull paradigm for data retrieval, but the actual reading from the data sources is done with data fetchers.
- **Data fetchers** are responsible for transferring data from data sources to the data management system. These fetchers may be called both by the data management system in a pull approach and by data sources in a push approach. The fetchers can add simple metadata and data links to the fetched data before it is stored as raw data. As an example, a data fetcher for a content management system (CMS) could add to a commit data, metadata about the repository the commit belongs to as well as data links to the files that the commit changes. Since, each data source should have its own data fetcher, more complex data linking would be done either in a data adapter or external analysis tool. For example, linking a commit to data from issue management system like a Jira would be done in the data adapter.
- **Raw data** represents semantically unmodified data as it is received from tools. Using raw data instead of pre-filtered data provides better support for advanced visualization operations, such as zooming in, stated in business requirements. Additionally, this approach allows for data storage in the original tool itself.
  In addition to raw data that has been retrieved from data sources, the data management system should store: (1) derived data that is generated by analysis tools, (2) additional data entered by users, and (3) metadata about the project.
- **Data adapters** implement a two-way adaptation. Firstly, it provides visualizations a uniform data model regardless of the data source. As an example, the data adapter may facilitate visualization for similar "ticket" data regardless of it being fetched from Jira or Trello. Secondly, data adapters may provide different data for different types of visualizations. A key part of the visualization-facing interface of data adapter is the data model it provides for the visualizations. Thus, “data model” is attached to the data adapter in Figure 2. Any data links that are not available in the raw data but are required

by the visualizations are created by the data adapters. This could for example include linking a commit from CMS to a developer or to a ticket in an issue management system.

2.2 Visualizations

Conventional visualizations often present a single viewpoint to the underlying data. More advanced visualizations, such as the electrocardiogram (ECK/EKG), are pieces of code that define or create concrete visualizations from one or several data sources. These visualizations represent generic software components that are not limited to displaying data but may also include functionalities related to interactivity, for example zooming in on and investigating details, which allow the user to search for the root causes of various issues. Additionally, these visualizations can be configured for specific stakeholder needs and preferences, certain dashboard usage via for example certain color schemes, and for various logical data sources and adapters.

From a visualization perspective, the data-adapter approach offers a distinct advantage on flexibility by decoupling data models from data sources. The logical data sources may represent for example an issue management system without specifying a concrete tool like Jira, offered through the data-adapter – where the data originates from one or more data sources. New data sources can be added through the data management system, allowing visualizations to focus on visualizing logical data in a source-agnostic way. The generic data model also allows for advanced features, such as zooming in and out of data, as visualizations have access to all the needed data at once, with relations between data items already mapped and traversable through a single API.

From an architectural viewpoint, individual visualizations are treated as black boxes that each handle their own data retrieval, processing, and presentation. This approach lends itself well to features from the micro-frontend paradigm – the development of small, independent applications working together to create a larger frontend – where each team is responsible for the development of a micro-frontend from start to finish. Thus, each team controls all required code, the building and deploying of their micro-frontend, and are not in any way reliant on the deployment schedule of other teams and their micro-frontends. Each team can thus also choose the technologies they need for their specific micro-frontend. However, each implementation of the reference architecture must weigh how heavily they invest towards the micro-frontend pattern as separating development activities with a micro-frontend-oriented approach may impose bottlenecks for data management.

2.3 Dashboards

The dashboards designed for different roles, needs and stakeholders are a key concept in VISDOM. A dashboard consists of one or more visualizations that together constitute a view that answers the visualization needs of the user. The proposed dashboard solution must be generic enough to support all activities in DevOps development in both stakeholder and company specific dashboards. This can be achieved by decoupling visualization logic from the dashboard itself. This approach provides the benefit of allowing dashboards to suit all stakeholder needs without imposing limitations on what is being visualized and allows customization for both stakeholder and company specific dashboards. In addition, this approach supports both research and development activities by providing a common platform for deploying visualizations between development teams in an agile way.

The dashboard design is shown in Figure 3. The yellow boxes represent the previously described data management system, passing data to visualizations marked in red. Individual visualizations are implemented as micro-frontends by various teams, organizations and other entities interested in specific visualization concepts or domains. The dashboard composer in blue allows for registering micro-frontends to be used within the dashboard. The dashboard composer allows users to select and layout one or more visualizations into views within said dashboard, based on the user's needs. Furthermore, the dashboard composer allows for creating default views for different roles based on the registered visualizations.

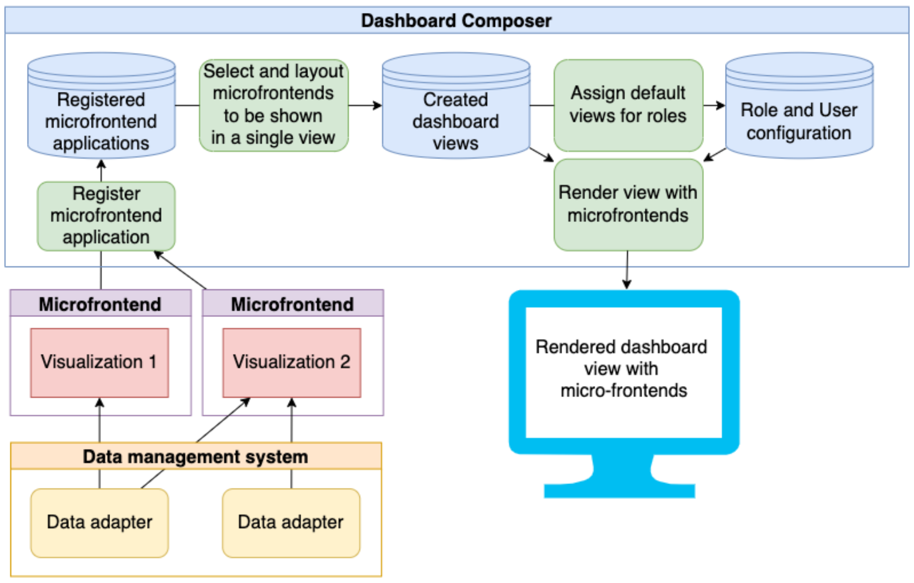

Dashboard Composer
Registered microfrontend applications
Select and layout microfrontends to be shown in a single view
Created dashboard views
Assign default views for roles
Role and User configuration
Register microfrontend application
Render view with microfrontends
Microfrontend
Visualization 1
Microfrontend
Visualization 2
Rendered dashboard view with micro-frontends
Data management system
Data adapter
Data adapter

Figure 3 The dashboard composer architecture.

The dashboard composer (Figure 3, in blue) is an integral component of the proposed configurable dashboard. It facilitates the creation, management, and rendering of individual view configurations, and houses the logic used in selecting views and visualization perspectives for different roles. In addition, the composer implements the necessary functionality for producing a coherent layout for the dashboard in terms of laying out individual visualizations on a grid and providing a responsive display for screens of varying size. The dashboard composer consists of the following subcomponents:

- **Registered micro-frontend applications** are stored in the composer as a basis for creating varying views in the dashboard. Each micro-frontend is a self-contained application that runs within the dashboard and includes one or more visualizations. The micro-frontends are made by teams, organizations or other entities that focus on visualizing a specific concept or domain. Each entity registers their available micro-frontends to the dashboard composer, which then allows views to be constructed from one or more micro-frontends. Furthermore, each micro-frontend implements its own visualization control logic, providing the necessary functionality for inter-visualization communication and relaying of events, such as changes in visualization perspective, to all the visualizations within each micro-frontend.
- **Created dashboard views** are stored in the composer and represent groups of visualizations for a specific interest, task, or other suitable concept for building individual dashboard views. In essence, view information includes a selection of micro-frontends and layout information. These configurations are then rendered as individual dashboard views. Users may also configure new views to their dashboards based on their interests and tasks.

- **Role and User configurations** represent default views created for specific roles and the personally customized views of individual users. The configurations are stored in the composer.

2.4 Security and privacy concerns

Security and privacy are essential aspects for architectural consideration as the proposed system might allow unethical or illegal tracking of stakeholders and other malicious activities if left unchecked. Especially, when dealing with any kind of personal data compliance with GDPR is compulsory in most cases. Features such as access control might be omitted for research prototypes but should still be considered in the overarching architecture. The key aspects for further consideration include access control in the form of authentication and authorization, and anonymization of data. These issues present a need for some kind of access federation system that provides a cross-domain solution for bridging access rights. Alternatively, future versions of the architecture may offer alternative solutions for working around this issue by deploying multiple data management systems for specific uses. This approach will depend on the degree of centralization for each implementation of the reference architecture.

This leads to the following requirements:

- The data management system, including both data fetchers and adapters, shall include an anonymizing service.
- The data management system shall include a user/access-token federation system.
- The data adapters need to implement some kind of "cross-domain" solution that bridges access rights.

3 Standardizable components and interfaces

This section summarizes both the exploitable software components that can be utilized either independently or together, and the standardizable components that implements relevant standards for industry-scale interoperability. The previously introduced data fetchers, adapters, and visualizations form a baseline selection of the exploitable software components. Each of these components must be addressed based on their structure, general behavior, and interfaces in order to present a unified architectural solution. This architectural proposal is aimed to support both research and development in the VISDOM project.

Data fetchers represent a link between the VISDOM data platform and external data sources. The fetchers present an opportunity for creating a standardized software component where the data fetching strategies are implemented in a way that a) allows the creation of new data fetchers for tool vendors and independent collaborators, and b) supports the underlying data management architecture with a clear interface towards VISDOM elements.
There are still challenges for the reusability:

- The design of adapter includes specifying the strategy on when and how data fetching occurs and whether caching is implemented for performance reasons. These decisions should be compatible with the target environment.
- The data fetcher can be re-usable only if the designs of data storages are compatible.

Data adapters are responsible for providing visualizations a uniform data model regardless of the data source and for providing different data for varying types of visualizations. Data adapters are the key in combining data from several sources for VISDOM related visualizations. Thus, the adapters present a suitable opportunity for standardization on both interface and component levels. The interface for providing visualizations a uniform data model regardless of the data source may be specified in a form of a generic description that aids implementation and integration efforts.

On a component level, the component's internal behavior may be suitable for standardization and reuse, as it operates towards both raw data access and visualization related logic. Reusability of the data adapter depends on the compatibility of the data storages of the systems.

Visualizations represent a more free-form selection of software components that may greatly vary between implementations depending on their intentions. However, each of the visualizations must still adhere to a common specification that allows the dashboard composer to register and display them within the dashboard. Following this architecture document, visualizations are independent components within the dashboard, and responsible for their own data querying and processing.

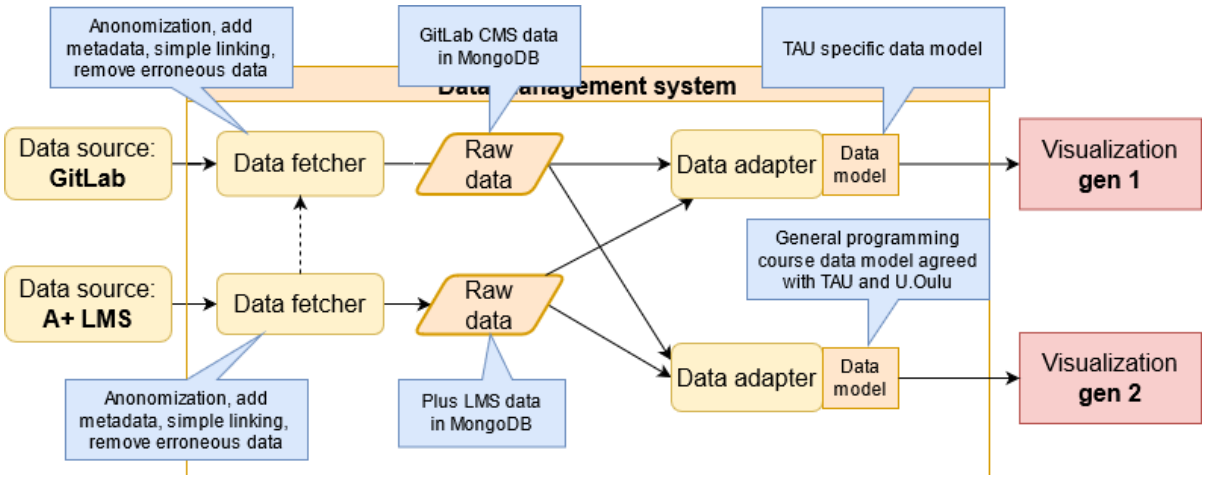
4 Analysis

In the section 4.1 different demo cases from the VISDOM project are analysed and mapped to the general reference architecture. Some of the demos match the reference architecture more closely than others but all have at least some parts that can be mapped to a component defined in the reference architecture. Some of the learnings as well as summarizing the observed similarities and differences from the demo cases are gathered in the section 4.2. Topics and ideas for future research are introduced in the section 4.3.

4.1 Reference implementations in the pilots and demos

During last months of the project, we checked the demonstrations to see how closely they follow the reference architecture. Some of the differences are due to implementation constraints: the reused components frameworks did not follow the VISDOM reference architecture. In addition, the implementation of the visualizations provided lessons learned and led to changes in the reference architecture. Below, we analyze the demos from the viewpoint of reference architecture. Each analysed case contains diagram where the components of the demonstration are mapped to the reference architecture.

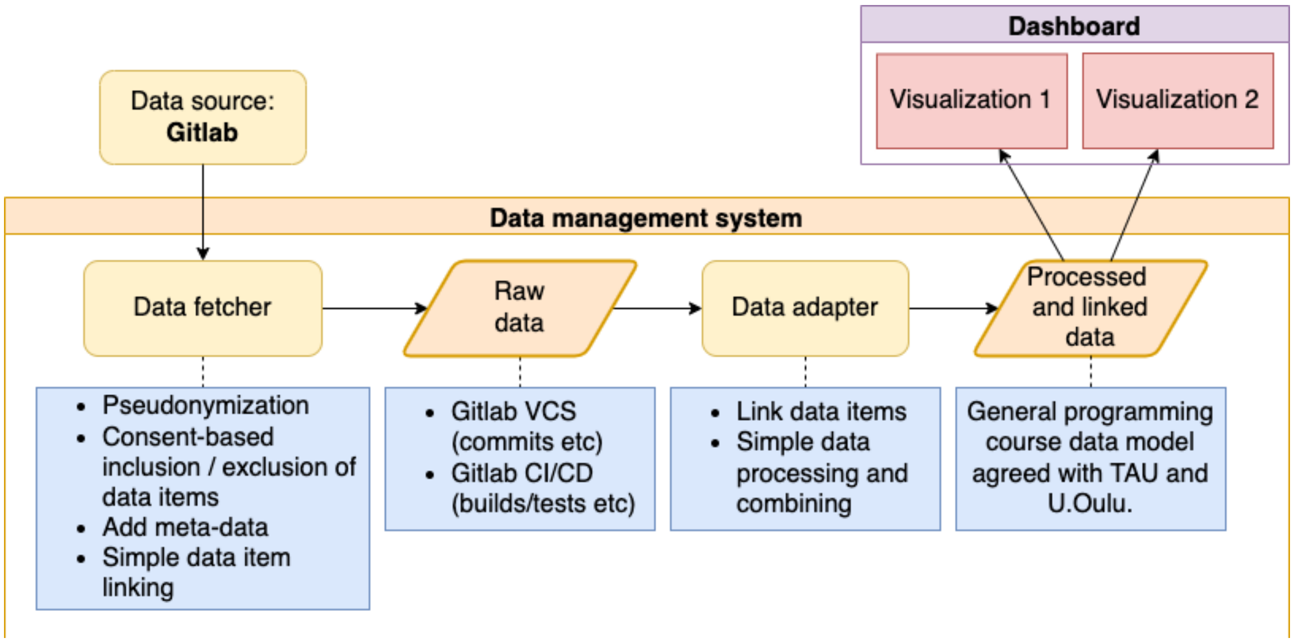
4.1.1 TAU teaching demo



- **Data sources.** The data sources are A+ Learning Management System[2] and GitLab.
- **Data fetchers.** A separate data fetcher for both data sources has been implemented. The data fetchers implement anonymization of the data, removal very obviously erroneous data, add some metadata (e.g., basic information about the course in A+ or GitLab project), and simple data linking. The data fetching is based on pull paradigm and the fetching for the A+ data is triggered by a periodic cron job. The fetching for the required GitLab data is triggered by the A+ data fetcher.
- **Data storage.** The data is stored in a Mongo database and is raw in a sense that is not tuned for any specific visualization.
- **Data adapters.** The data adapters are implemented with Apache Spark. The TAU implementation has two generations: the first provides the data as a single and big document containing a lot of data. The second generation aimed at a common data model for Tampere and Oulu Universities. The second-generation adapter provides an API that allows the visualization to query the data it needs. The TAU adapters implement some linking of the data items. The results from the adapter are cached to the data storage, so that the mapping from the raw data to the data model needs to be done only once whenever the raw data changes.
- No advanced system for managing the credential to data sources was implemented; the username and required tokens are stored in a configuration file. Access to end-users of the teaching dashboards is password-protected.
- The architecture in the TAU teaching demo was purposefully done to match the reference architecture defined D2.5.1 as closely as was deemed reasonable in order to get practical knowledge about the viability of the architecture in practice.

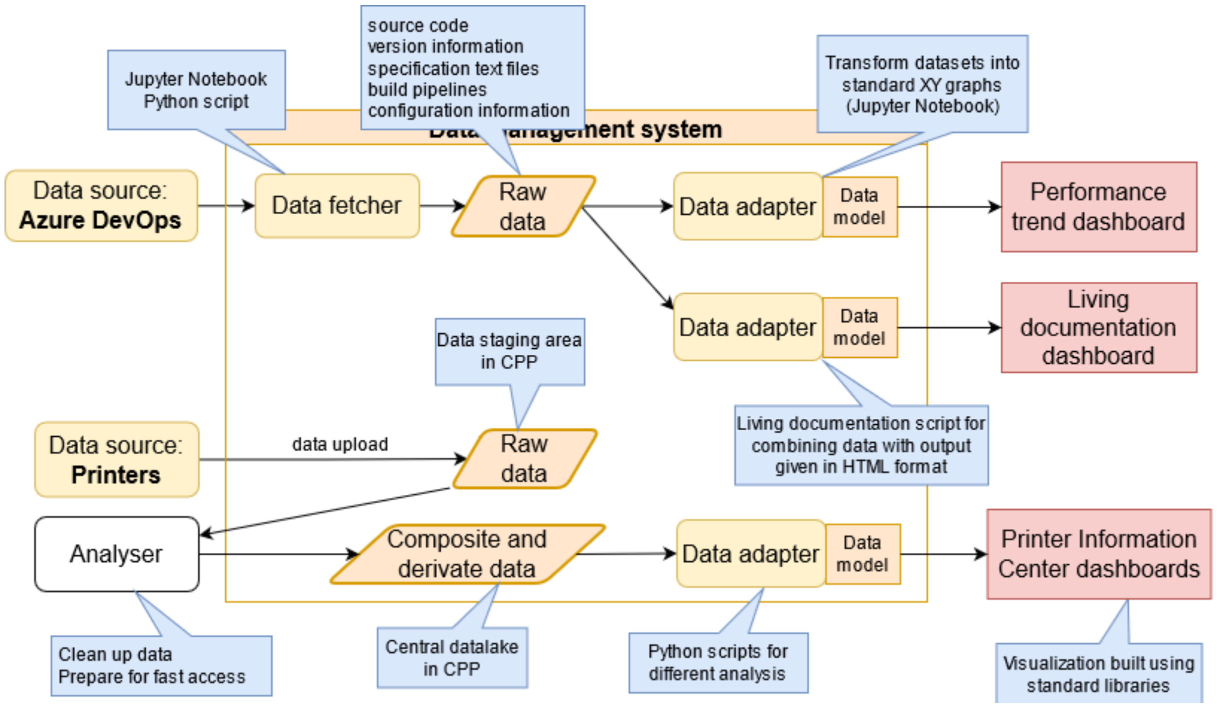
[2] https://apluslms.github.io/

4.1.2 Oulu teaching demo


Dashboard
Visualization 1
Visualization 2
Data source: Gitlab
Data management system
Data fetcher
Raw data
Data adapter
Processed and linked data
Pseudonymization
Consent-based inclusion / exclusion of data items
Add meta-data
Simple data item linking
Gitlab VCS (commits etc)
Gitlab CI/CD (builds/tests etc)
Link data items
Simple data processing and combining
General programming course data model agreed with TAU and U.Oulu.

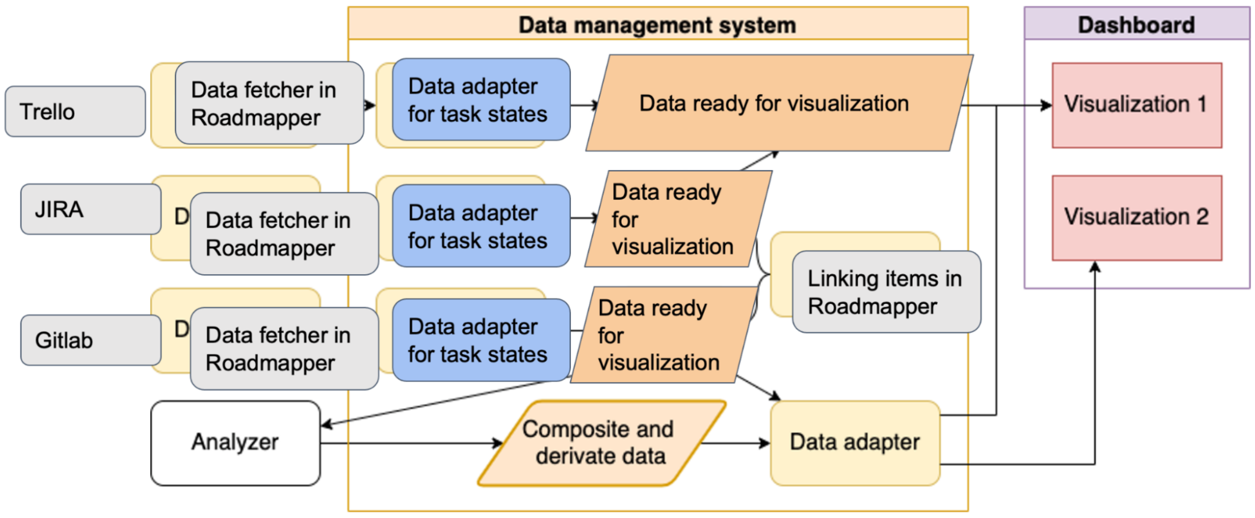
- University of Oulu used the same data backend as TAU teaching demo and thus also match closely to the defined reference architecture.
- **Data sources.** GitLab (commit and CI/CD pipeline data).
- **Data fetcher.** Uses the same GitLab data fetcher as TAU teaching demo. Externally managed so that only data from those students who have given their consent is fetched.
- **Data storage.** The data is stored in a Mongo database and is raw in a sense that is not tuned for any specific visualization.
- **Data adapter.** Uses the second-generation adapter from TAU teaching demo that provides the data in a general data model agreed between the universities containing links between the connected data items.

4.1.3 Canon quality demo


source code
version information
specification text files
build pipelines
configuration information
Jupyter Notebook
Python script
Transform datasets into standard XY graphs (Jupyter Notebook)
Data management system
Data source: Azure DevOps
Data fetcher
Raw data
Data adapter
Data model
Performance trend dashboard
Data adapter
Data model
Living documentation dashboard
Data staging area in CPP
Living documentation script for combining data with output given in HTML format
Data source: Printers
data upload
Raw data
Analyser
Composite and derivate data
Data adapter
Data model
Printer Information Center dashboards
Clean up data
Prepare for fast access
Central datalake in CPP
Python scripts for different analysis
Visualization built using standard libraries

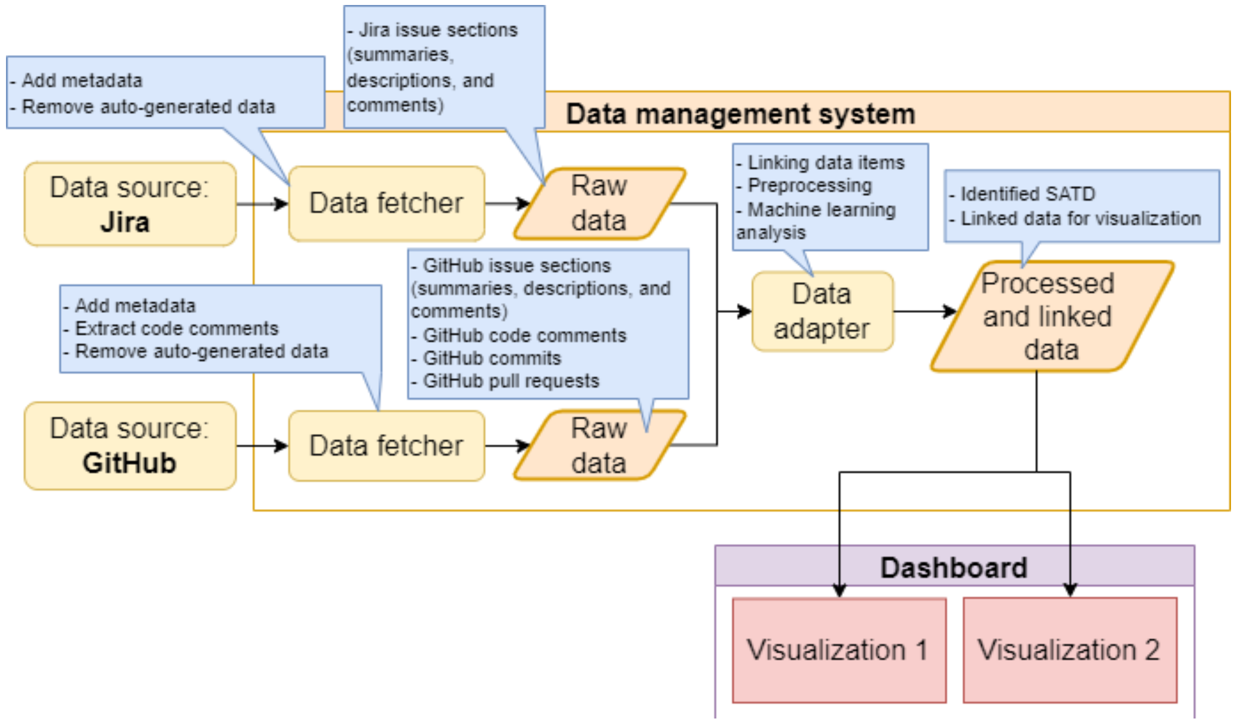
- Three different kinds of dashboards are considered here: performance trend and living documentation dashboards for data from Azure DevOps data and printer information center dashboards for data received from deployed printers related data.
- **Data sources.** For performance trend and living document dashboards raw data from Azure DevOps (source code, version information, specification text files, build pipelines) is used. Also, configuration information from the regression tests that have been executed by the build pipelines is included. For printer related dashboards, the raw data source is data generated by the printers deployed in the field. The printer data is cleaned up and prepared for faster access with a separate program.
- **Data fetchers.** Python scripts running on Jupyter Notebook are used to fetch the source code and other related data from Azure DevOps. The script can filter the data on range of build numbers and dates. The raw printer data is uploaded by the printers through network connection to Canon Production Printing (CPP) staging area.
- **Data storage.** The cleaned printer data is stored in a central datalake in CPP. No separate data storage is used for the data fetched from Azure DevOps.
- **Data adapters.** For performance trend dashboard, a Jupyter Notebook script is used to transform the fetched raw data to 2-dimensional graphs. For living documentation dashboard, a different script that combines the specification text files with the test results used and creates the dashboard is used. For the printer related dashboards separate Python scripts are used to transform the stored printer data for the visualizations in the dashboards.
- In the dashboards for Azure DevOps data there are separate scripts that can be mapped to the fetcher and adapter components in the reference architecture. However, in these cases the raw data is not stored separately but is instead directed straight to the adapter script and from there to the dashboards.
- The printer dashboard case follows a push paradigm where the deployed printers upload raw data to a staging area and the data is then cleaned up by a separate program. There is no separate fetcher component but other than that this case can be mapped to the reference architecture quite well.

4.1.4 Roadmapper tool from Vincit


Data management system
Dashboard
Trello
JIRA
Gitlab
Data fetcher in Roadmapper
Data fetcher in Roadmapper
Data fetcher in Roadmapper
Data adapter for task states
Data adapter for task states
Data adapter for task states
Data ready for visualization
Data ready for visualization
Data ready for visualization
Linking items in Roadmapper
Analyzer
Composite and derivate data
Data adapter
Visualization 1
Visualization 2

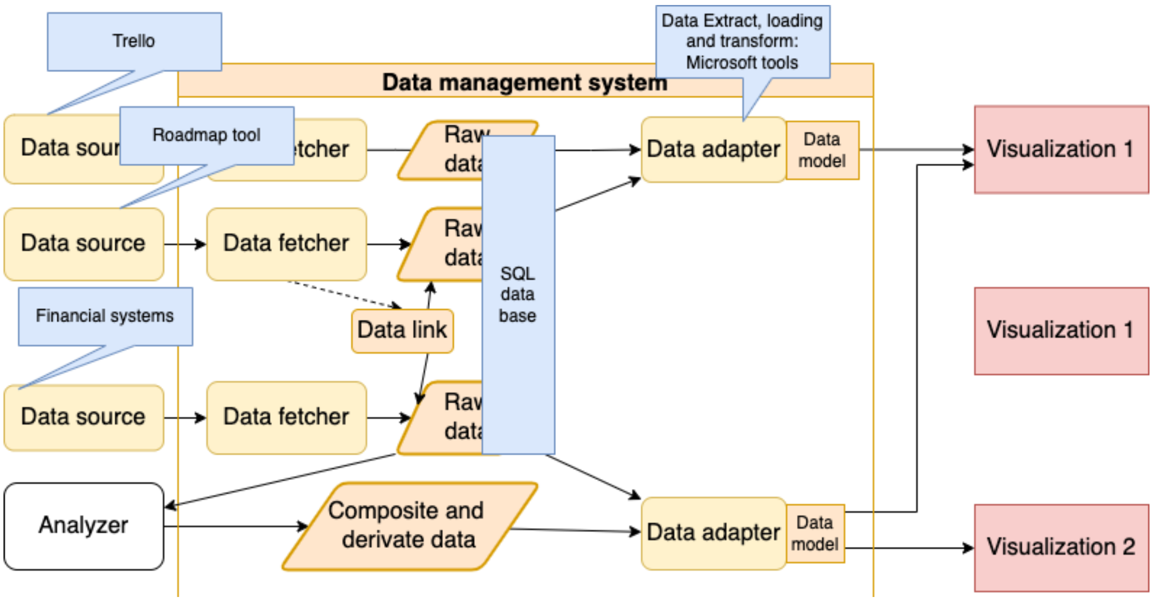
- **Data sources.** The supported data sources are Trello, JIRA, and GitLab (issue/task information). The data is complemented with information entered by the users. Access to data sources is based on tokens that the users of the tool provide.
- **Data fetchers.** A generic data fetching framework has special components for each of the supported data sources.
- **Data adapters.** The fetched raw data is transformed to a data model that can be used directly in the visualizations. Unlike in the reference architecture this data transformation is done before storing the data.
- **Data storage.** The data is stored in a PostgreSQL database and the used data model is tuned for the visualizations of the Roadmapper tool.
- In this use case storing only the visualization-ready data makes sense, since the backend is specific for the Roadmapper tool, and the stored data is only expected to be used within the tool itself. Other than that, the architecture used in the tool can be easily mapped to the reference architecture.

4.1.5 RUG tool to analyze self-admitted technical dept



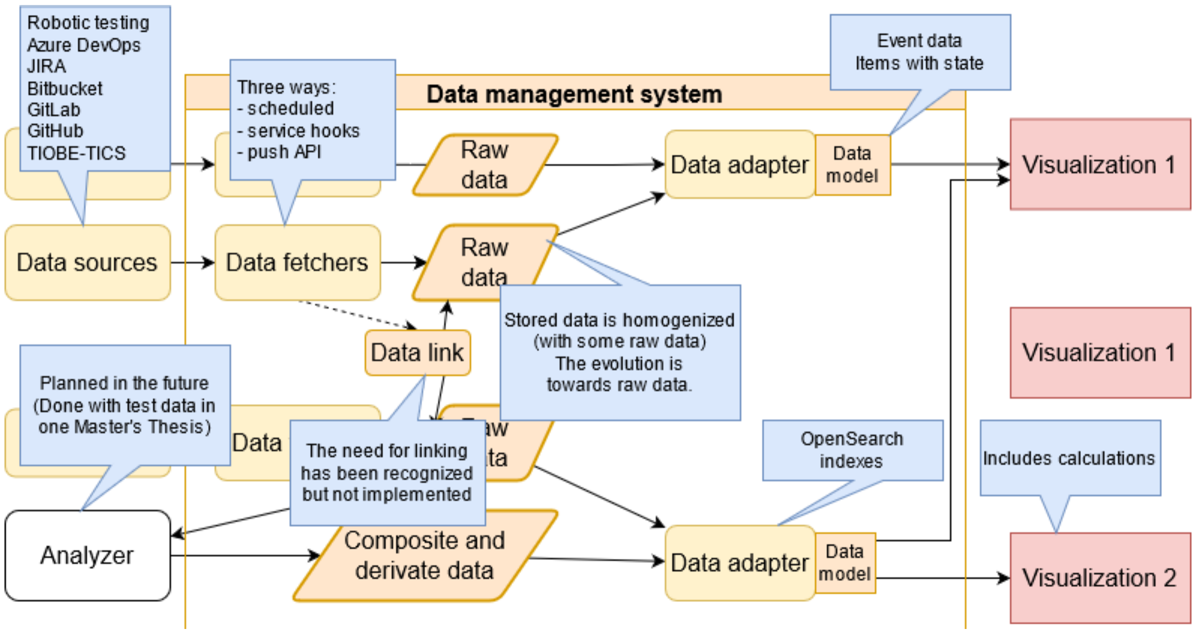
- **Data sources.** The data sources are GitHub and Jira issue tracking system.
- **Data fetchers.** There are two data fetchers: one collects code comments, commit messages, pull requests, and GitHub issues from GitHub, while the other one gathers Jira issues from Jira issue tracking systems. Some auto-generated data is removed by the fetchers. For example, comments about the static code analysis results that are added by bots are removed.
- **Data adapters.** The data adapter links the data from different sources (i.e., code comments, commit messages, issues, and pull requests), preprocesses the data, and identifies self-admitted technical debt (SATD) from the processed data.
- **Data storage.** The data is saved in a PostgreSQL database.
- This demo case follows the reference architecture quite closely. Some raw data cleaning and adding of metadata is done in the data fetchers, similarly to the TAU demo case. In this case the data adapter does machine learning analysis and the results are stored to the data storage for the use of the visualizations. The explicitly marked result data storing can be considered to be similar to a cache in the TAU demo case adapters.

4.1.6 Invenco financial value dashboard



- Quality/process management dashboard with focus on visualization of the financial value of features.
- **Data sources.** Trello, Roadmap tool, and financial systems.
- **Data fetchers** and **adapters.** Microsoft Business Intelligence tools are used to fetch the raw data from the data sources and to transform the fetched data so that it can be used in the visualizations. Access control is also handled by these Microsoft BI tools.
- **Data storage.** The data is stored in a Microsoft SQL Server database.
- In this case, a set of commercial tools are used to handle the data management all the way from the data sources to the visualizations. Without further research it is not exactly clear how the Microsoft BI tools components should be mapped to the reference architecture. However, there are still components that handle the data fetching from the data sources, i.e. data fetchers, and components that map and combine the fetched data to the data models used in the visualizations, i.e. data adapters, that should correspond the components in the reference architecture.

4.1.7 Qentinel/Copado Robotic Testing dashboard



- **Data sources.** Several different data sources are included: Azure DevOps, JIRA, Bitbucket, GitLab, GitHub, TIOBE-TICS, and robotic testing.
- **Data fetchers.** There are different ways to fetch the data from the sources. Some data fetching is done in pull approach using scheduling while other data is received through push API or by receiving a notification through a service hook.
- **Data storage.** The fetched data is stored mostly in a homogenized format to OpenSearch cluster. Some part of the data is also stored as raw data. The evolution of the system has been that increased amount of the fetched raw data is stored as it is. The creation of links between the data items has been recognized as something that would be helpful but has so far not been implemented. Also, there are plans to add separate data analyzing components to provide additional information.
- **Data adapters.** OpenSearch has been found out to be a good solution for getting the data to the visualizations. Some of the required calculations is done in the visualization code.
- Overall, the architecture in this case has evolved towards the reference architecture, i.e. storing as much of the raw data as possible, linking data items, etc., and the future plans are also to continue to that direction.

4.2 Lessons learned

Several topics were learned during prototype implementations of the proposed architecture. The learnings are divided into three sections: 4.2.1 for topics related to the data handling and data storage, 4.2.2 for topics related to various performance issues discovered with showing the data in the visualizations, and 4.2.3 for topics related to various requirements that were discovered regarding the visualizations.

4.2.1 Data handling and data storage

- In many cases anonymization of data and removal of access to forbidden data needs to be solved. In some cases, it may be necessary to anonymize the data before storing it into raw data. Thus, anonymization must be done (at the latest) in the data fetcher. In some cases, the stored data can contain user identification. This decision does not depend on the technical constraints but is largely determined by legal and organizational constraints. For example, in the teaching use case, student identification cannot be stored outsize teaching systems. Also, any data that the data fetcher might have access to but that cannot be used at all outside the original data source due to for example GDPR concerns should be fully filtered out by the fetcher.
    - It should be noted that pseudonymization instead of full anonymization can be used in certain situations, and it allows linking related raw data items whereas full anonymization might not allow the data linking.
- Erroneous data can be removed by both the fetcher and the adapter. Clearly erroneous data should be discarded by the fetcher, but the adapter that knows about the use of the data can deal with more complex issues like missing values, wrong data types (e.g. strings instead of integers), or values out of valid ranges (e.g. expected integer between 0 and 10 but got a negative value instead).
- The original architecture (D2.5.1) included a “Data Linker” component. A separate linking component doesn’t seem to be useful. Seems, that linking is done in other components (fetcher, adapter, visualization). The fetchers can also work as data linkers to connect semantically linked data together for cases where the linking is simple and direct. For example, linking connected commits and files from the same CMS can naturally be done by the fetchers. The data adapter on the other hand can do linking that is relevant for the visualizations it serves.
- The reference architecture presents the idea that the full raw data from the data sources should be stored in order to allow the visualizations to have access to the details whenever necessary. Additionally, some processed or derived data can be saved to complement raw data. Depending on the use case this approach of storing the full raw data might seem overly heavy adding unnecessary complexity to the architecture. However, unless the use cases for the data are fully known with no possibility that they will change, any processing that is done to the raw data before storing it might remove details that would be useful in the future.
    - Naturally some data processing can still be done to the raw data. For example, if it is fully known that some details included in the raw data will never be used, those details can be removed before the fetched raw data is stored. As an example, if the raw data contains the full list of line changes for a Git commit but at most only the number of added or deleted lines might be used, the full change log can be removed by the data fetcher.
    - On the other hand, if the use cases for the data are fully known, it can be reasonable to process the data first before putting it into the data storage. As an example, in the Roadmapper tool (4.1.4) the raw data is processed to the used visualization-ready data model before storing it and at least so far this approach has worked well. It should be noted however, that only storing the data in a processed data model will limit the options for any future development.

- If the used dataset is small enough, or there are issues (e.g. legal issues) related to storing the fetched data, the separate data storage can be left out of the implemented architecture. I.e., have only the data fetcher for fetching the data, and data adapter to transform the fetched data for the visualization. Depending on the use case, this approach might not be feasible. For example, if the data fetching for the dataset takes too long or the data processing part is too complex to be handled in memory only.

- If the data sources provide data that is not publicly available, then care regarding the credentials and access control to the fetched data must be taken in order to limit the access to the data to only those users who are allowed to access the stored data.

4.2.2 Performance related learnings

- On the visualization side, if there is a large amount of data the visualization needs to load at the initialization, loading the needed data sequentially can lead to slow downs and poor user experience. Use of concurrency and especially an optimized use of the relevant adapter is needed in those cases.
- On the adapter side, a proper implementation of the visualization interface of the adapter is critical for the performance. If the visualization receives too much data, the communication delays and preprocessing in the client causes delays. On the other hand, a need for too many requests also slows down the system. Based on the experiment in the pilot (especially the TAU teaching demo), API that has some kind of Query language support that allows filtering and limiting the output, instead of pure and simple REST API should be used.
- We also learned that some kind of cache for the adapter results is very likely required in most use cases. In the teaching demo case, the data prepared by the adapter is cached in order to provide the data efficiently to the subsequent visualizations. The adapter cache is updated whenever a query is made to the adapter and the cache is out-of-date, i.e., the raw data has been updated. This approach worked well in the teaching demo case since the raw data was updated with scheduled data fetching and thus the cache could be updated immediately after the raw data had been updated. Another example where a cache is almost certainly required is the self-admitted technical dept case (4.1.5) where the adapter runs machine learning analysis for the raw data which will likely take more time than what the user of the visualization can be expected to wait.
- Some of the performance issues are possible to solve by using ready-made tools. For example, in the Robotic Testing demo (4.1.7) using the indexing and search capabilities of OpenSearch was found to be a good solution. Also, some of the issues can be avoided by using a fully commercial utilities like Microsoft BI but further study is required to be able to present any further information.

4.2.3 Requirements regarding visualizations

- Visualizations that match with the reference architecture have both client and server-side functionality and should co-exist within a single dashboard view. Thus, some kind of micro-frontend architecture is needed. The prototype implementations of TAU and University of Oulu uses the Single-SPA framework[3]. The prototype implementation revealed challenges[4]:

- The users may want ways to organize the layout according to their needs. The current prototype implementation does provide this functionality for the end-users.
- Sharing of access tokens or authentication credentials for each micro-frontend is challenging without a dedicated system for their management.

- There needs to be a possibility to synchronize the visualizations, e.g., so that changing the time frame of focus in one visualization is repeated in other visualizations in the same dashboard. Some, publish-subscribe solution is needed (The TAU+OU prototype used MQTT). However, use of this kind of communication mechanism may complicate the architecture and reduce re-usability of the visualization. The synchronization of visualization is not only an implementation issue: the exact needs are non-trivial, too.
- Users need a way to add new visualizations, e.g. to compare two projects (in teaching case two students). The current prototype implementation does provide this functionality for the end-users.

4.3 Future work

Most of the demos worked inside a single “trust domain” and thus there was no need to implement and evaluate the security and privacy-related requirements. In TAU teaching demo, the student information was anonymized and access to the running visualizations was restricted to the relevant personnel. Properly defining and evaluating these requirements is left for future research.

A mixture of push and pull approaches increases complexity, which in turn sets a requirement to carefully design update cycles. The performance, synchronization of data, and possible real-time properties like in real-time BI[5] need to be considered. While a couple of the demonstrations included data fetching using not just pull approach but also push approach (4.1.3 and 4.1.7), the way the visualization-friendly data was passed on to the implemented dashboards was still done using only pull-type of fetching from the adapter. Thus, the full handling of the different paradigms is left for future research.

It is also left for further study on how the presented reference architecture relate to commercial products like Microsoft Business Intelligence tools. And also, on how standards like OSLC[6] could be used with the reference architecture.

The original architecture (D2.5.1) included “Data broker” component which would have been used by visualizations to find out the available data and direct the visualization to the correct data adapter. This concept was not used in any of the demonstrations and thus further study is required in order to check if this kind of data/adapter discover mechanism is feasible.

[3] https://single-spa.js.org

[4] Thuy Phuong Nhi Tran: Componentized Visualization ComponentsWithMicro-Frontends, BSc thesis, Tampere University, 2022.

[5] See e.g., https://en.wikipedia.org/wiki/Real-time_business_intelligence
[6] https://open-services.net/

5 Conclusions

Based on the analyzed demonstrations the high-level reference architecture presented in this documented is feasible and can be used to implement a system where data collected from various data sources is processed and then visualized to the user. The concept of storing as much as possible of the raw data gathered from the data sources allows the visualizations access to the bottom-level details when necessary. Also, it allows updating the dashboards with new visualizations that use different set of attributes from the data sources without touching the actual data fetching process itself.

However, it should be noted that depending on the actual use cases, a more straightforward approach (fetch data, process the data, possible store the processed data in a cache, visualize the processed data) can be sensible and that the full reference architecture is not to be implemented blindly in every use case involving data visualization. All in all, some experiences have shown that a system that continues to be updated and improved tend to move from the straightforward approach to the concepts presented in the reference architecture.

There are still some open questions about the architecture, especially on how the security and privacy related issues should be handled, as well as on how the presented reference architecture relates to various commercial products. Also, how the solutions presented in various data management standards could be used regarding systems implemented following the reference architecture requires some further study.