UNIVERSITY OF TARTU
Institute of Computer Science
Data Science Curriculum

Mait Metelitsa

A Functional Prototype and General Architecture of Analytic Data Management for a Railway Company

Master's Thesis (15 ECTS)

Supervisor(s): Kristo Raun, MSc
Prof. Ahmed Awad, PhD

Tartu 2024

A Functional Prototype and General Architecture of Analytic Data Management for a Railway Company

Abstract:

The thesis delves into the realm of data engineering and business intelligence within the context of a railway company. The work aims to address the challenges posed by the modernisation efforts of EVR's analytic data management platform.

One of the key novel aspects of the thesis lies in its transition from the AS-IS state to the TO-BE state of analytic data management in EVR. This transition involves a detailed analysis of decision dimensions and technology alternatives for the future architecture of EVR's data management system. The proposed TO-BE architecture advocates for a flexible hybrid approach combining on-premises/Infrastructure as a Service (IaaS) and Software as a Service (SaaS) solutions tailored to the type and complexity of data sources.

Furthermore, the thesis presents practical use cases based on the TO-BE architecture, showcasing the implementation of end-to-end analyses of purchase invoices and railway level crossings' log data. By integrating technologies such as ETL processes, Dagster for data orchestration, Postgres for data storage, Streamlit for data visualisation, and XML-fetching for data retrieval, the thesis demonstrates a tangible contribution towards further improving EVR's data engineering capabilities and preparing the company for the adoption of data lakehouse SaaS platform.

Overall, the thesis contributes to the field of data management by providing a structured framework for decision-making in analytic data platform architectural design, emphasising the importance of technology choices and trade-offs in optimising analytic data management for a complex organisation like EVR.

Given the limited exploration of data engineering and business intelligence within the railway sector, this thesis fills a significant knowledge gap by providing insights and recommendations tailored to the unique requirements and challenges of managing data in a railway company undergoing analytics platform modernisation.

Keywords:

ETL, data lakehouse, Dagster, Postgres, Streamlit, XML-fetching, railway sector, log-based data mining, predictive maintenance, platform requirements engineering

CERCS:

P170 Computer science, numerical analysis, systems, control

P175 Informatics, systems theory

A Functional Prototype and General Architecture of Analytic Data Management for a Railway Company

Mait Metelitsa

Data Science (MSc) 2024

Institute of Computer Science

University of Tartu

Supervisors: Kristo Raun, MSc;
Prof. Ahmed Awad, PhD

#UniTartuCS

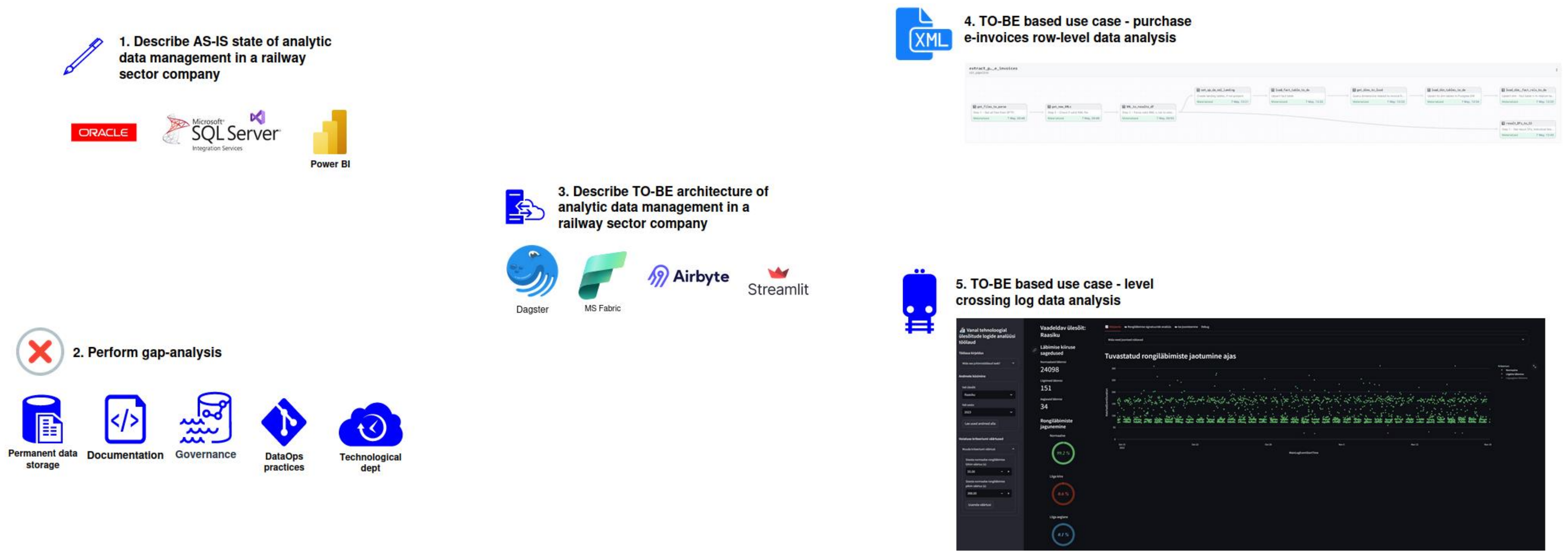

1. Describe AS-IS state of analytic data management in a railway sector company
ORACLE
Microsoft SQL Server Integration Services
Power BI
2. Perform gap-analysis
Permanent data storage
Documentation
Governance
DataOps practices
Technological dept
3. Describe TO-BE architecture of analytic data management in a railway sector company
Dagster
MS Fabric
Airbyte
Streamlit
4. TO-BE based use case - purchase e-invoices row-level data analysis
XML
extract_p_e_invoices
5. TO-BE based use case - level crossing log data analysis
Vanal tehnoloogial ülesõitude logide analüüsi töölaud
Vaadeldav ülesõit: Raasiku
Läbimise kiiruse sagedused
24098
151
34
Rongiläbimiste jagunemine
Tuvastatud rongiläbimiste jaotumine ajas

Figure 1. Visual abstract.

Analüütilise andmehalduse funktsionaalne prototüüp ja üldine tehniline arhitektuur raudtee-ettevõtte jaoks

Lühikokkuvõte:

Magistritöö käsitleb andmetehnika ja ärianalüütika temaatikat raudtee-ettevõttes. Töö eesmärk on lahendada EVR-i analüütilise andmehaldusplatvormi moderniseerimisega seotud väljakutseid.

Lõputöö üks peamisi uudseid külgi on detailne analüüsipõhine kirjeldust sellest, kuidas EVR-il oleks kõige otstarbekam nii rahaliselt kui ka arhitektuuriliselt oma analüütilist andmehaldusplatvormi moderniseerida. Töös on avatud hetkeolukord, kirjeldatud lähteandmete nomenklatuuri ning läbi otsustusdimensioonide ja tehnoloogiliste alternatiivide üksikasjaliku analüüsi on jõutud tulevase arhitektuuri teekaardini.

Väljapakutud tulevikuarhitektuur toetab paindlikku hübriidset lähenemisviisi, mis ühendab infrastruktuuri kui teenust (IaaS) ja tarkvara kui teenust (SaaS) käitumise mudeleid. Rõhutamist väärib, et tasakaal IaaS ja SaaS tarkvara käitamismudelite vahel on kohandatud vastavalt andmeallikate tüübile ja keerukusele.

Praktilise panusena realiseeritakse töös tulevikuarhitektuuri komponentidele tuginedes kaks kaasust. Esmalt, ostuarvete XML-ide ridade ning teiseks, ülesõitude logiandmete automatiseeritud analüüsi- ja andmetöötlusvood. Praktiliste kaasuste jaoks arendatud lahendused baseeruvad mh tehnoloogiatel nagu Dagster andmetöötlusvoogude orkestreerimiseks, Postgres andmete säilitamiseks ja Streamlit andmete visualiseerimiseks. Sellega annab lõputöö praktilise panuse EVRi andmetöötlusvõimekuse edasisse parandamisse ja valmistab ette tehnilist võimekust ja asutusesisest kompetentsi SaaS andmejärve platvormi kasutuselevõtuks. Eeldatavalt saab selleks platvormiks olema Microsoft Fabric.

Lõputöö annab üldisema panuse andmehalduse ja andmetehnika valdkonda, pakkudes struktureeritud raamistikku analüütiliste lahenduste arhitektuuridisaini loomiseks. Võttes arvesse andmetehnika ja ärianalüüsi suhteliselt mõõdukat varasemat vaagimist raudteesektoris, toetab käesolev lõputöö märkimisväärselt selle teadmislünga ületamist.

Võtmesõnad:

ETL, andmejärv, Dagster, Postgres, Streamlit, XML, raudteesektor, logiandmete kaeve, ennetav hooldus, analüütika platvormi nõuete analüüs

CERCS:

P170 Arvutiteadus, numbriline analüüs, süsteemid, kontroll

P175 Informaatika, süsteemide teooria

Analüütilise andmehalduse funktsionaalne prototüüp ja üldine tehniline arhitektuur raudtee-ettevõtte jaoks

Mait Metelitsa

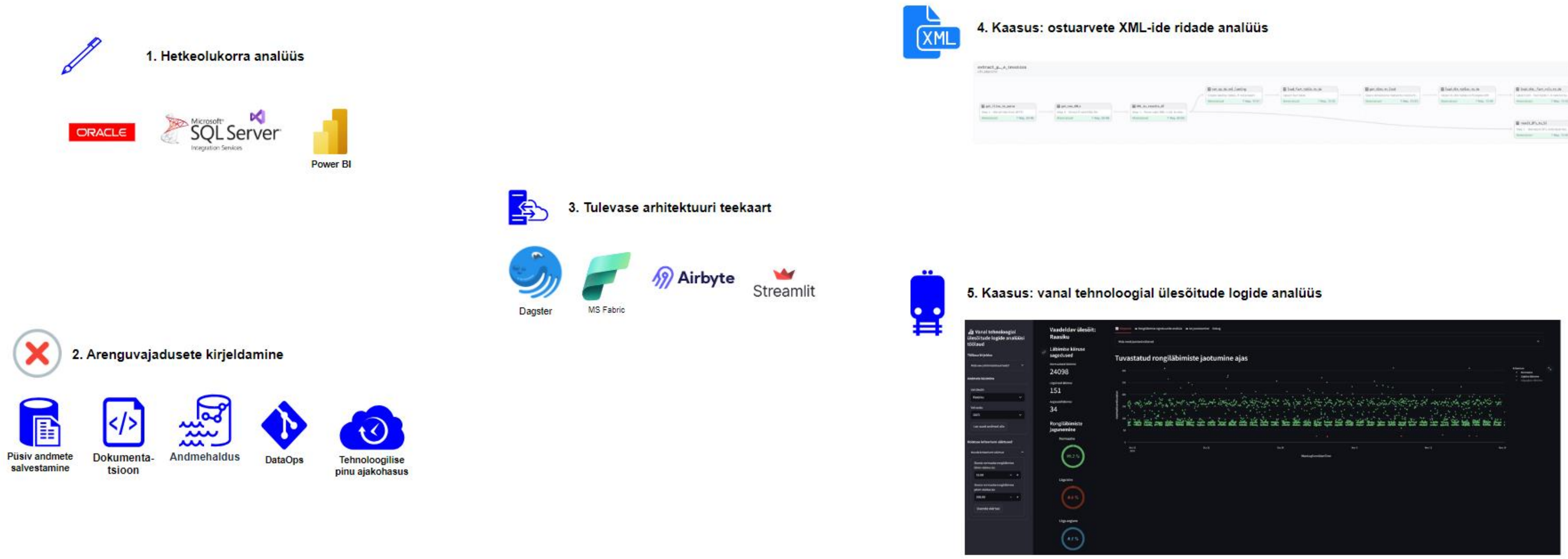
Andmeteadus (MSc) 2024

Arvutiteaduste Instituut

Tartu Ülikool

Juhendaja: Kristo Raun, MSc;
Prof. Ahmed Awad, PhD

#UniTartuCS


1. Hetkeolukorra analüüs
ORACLE
Microsoft SQL Server Integration Services
Power BI
2. Arenguvajadusete kirjeldamine
Püsiv andmete salvestamine
Dokumenta-tsioon
Andmehaldus
DataOps
Tehnoloogilise pinu ajakohasus
3. Tulevase arhitektuuri teekaart
Dagster
MS Fabric
Airbyte
Streamlit
4. Kaasus: ostuarvete XML-ide ridade analüüs
XML
5. Kaasus: vanal tehnoloogial ülesõitude logide analüüs
Vaadeldav ülesõit: Raasiku
Läbimise kiiruse sagedused
24098
151
34
Tuvastatud rongiläbimiste jaotumine ajas

Joonis 2. Visuaalne kokkuvõte.

Table of Contents

1 Introduction

Estonian Railways Limited (EVR) is a government-owned company whose core business is building, operating and maintaining Estonian railway infrastructure. Per the European Union Common Market legal framework, the same company cannot concordantly fulfil the role of railway undertaker and infrastructure manager in the interests of encouraging free market competition. Therefore, as an infrastructure manager, since 2001, EVR has not operated cargo trains.

The main strategic focus of EVR is the modernisation of its infrastructure, which entails electrification, digitalisation and boosting the maximum allowed speed of currently operated rail lines — the railway infrastructure reconstruction due to be completed by 2028 costs nearly 800 million Euros. Digitalisation will increase the variety of data sources, velocity, and overall amount of data. The change poses a challenge at the technical, work processes, and culture levels.

With a focus on business intelligence and analytic data, the current thesis tries to tackle the previously outlined challenges at the organisational and technical levels. In addition to giving a theory-based overview of the critical data engineering concepts, the thesis also describes the AS-IS state of business intelligence and analytic data management in EVR. A detailed description of the AS-IS state is essential because the thesis is dealing with a brownfield-type project where already-made decisions and sunken costs annulate some options in choosing the technological stack that would be reasonable in a clean-slate greenfield project.

Moving from general to specific, the following central part of the thesis focuses on TO-BE. The thesis outlines the main decision dimensions and technological trade-offs that general architectural design decisions must answer. TO-BE architectural decision dimensions are elaborated through relevant technology alternatives. The analytic part of the thesis concludes with a presentation of the general TO-BE architecture for EVR's analytic data management and business intelligence.

The last part of the thesis presents the practical implementation of two business-relevant use cases based on technologies specified in TO-BE business intelligence and analytic data management architecture. The first practically implemented use case consisted of end-to-end analysis of row-level data of purchase invoice XML files, including initial data-fetching and cleaning, dimensional normalisation to star schema, data enrichment and reporting from the Postgres-based data warehouse.

As a second use case, the author created an automatic data pipeline to analyse railway level crossings' log data from legacy relay-based level crossing systems to extract train pass-through events and respective timings. The goal was to create an exploratory tool which would allow for the monitoring of train-passing event timings by the level crossing, with particular emphasis being the detection of train passings with spuriously short duration, indicating an *in situ* malfunction of train detection. As train passing event signatures and normative train passing timings differ by the level crossing, the author expanded the data visualisation dashboard with minimal CRUD capability to save per-level crossing alert threshold values set manually by the end-user.

The practical work contributes significantly by preparing EVR for a Microsoft Fabric-based proof of concept study with Microsoft's Baltics cloud services partner company. The future proof of concept study will involve the same datasets and analytic questions covered in the thesis's two end-to-end practically implemented cases.

2 Theoretical Background

This chapter provides an overview of the data engineering challenges addressed by this thesis, both in terms of requirements engineering and practical implementation. Additionally, by summarising crucial data engineering concepts, this chapter lays the groundwork for part four of the thesis, which proposes future analytics data management architecture for EVR. Furthermore, as the railway sector as a problem domain is unique, the background section of the thesis also gives an overview of the most referenced review articles that deal with data engineering in the railway sector.

2.1 Methodological Background and Scope of the Thesis

In order to ensure the future architecture of EVR's analytic data management is adequately specified, the current thesis has established two key objectives. The first is to undertake requirements engineering, a crucial step often overlooked in corporate data engineering projects [1, 2]. This oversight can result in accidental complexity within the architecture, as the data engineering field is rife with competing technologies that offer similar functionality with differing underlying architecture, deployment patterns, and technical trade-offs [3, 4, 5]. As a result, the thesis places significant emphasis on describing the current state of analytic data management in EVR, including mapping and classifying the source data systems that are not utilised in the practical implementation of the thesis. Broader coverage of source data is needed because the future analytic data management platform towards which the gap-based requirement engineering and analysis contributes must serve all the future analytic needs of the EVR, which are much broader than the two practical cases solved by the current thesis.

Secondly, the thesis aims to provide the company with a comprehensive end-to-end analytic data management architecture, broadly covering everything from data ingestion to reporting. The task involves a detailed discussion of various technologies and platforms, some of which are not included in the current technology set used to realise the practical usage cases. The centrepiece of the proposed architecture is Microsoft's cloud-based SaaS[1] all-in-one analytic data management solution Fabric [6], which covers data engineering steps from data movement to data science and business intelligence. Detailed rationalisation for selecting Microsoft Fabric over its competitors is presented in latter chapters of the thesis. Although the implemented use cases do not utilise Fabric's technical capabilities, the rationale behind choosing this platform is thoroughly described due to its costliness and multifaceted capabilities. A detailed description of Microsoft Fabric is necessary to determine how to interface source data systems with the cloud platform (including thinking through network interfaces and authentication mechanisms) and which capabilities to implement based on Fabric's costly SaaS offering versus implementing required steps on cheaper on-prem or

[1] SaaS, or Software as a Service, is a cloud-based service model where software applications are hosted by a service provider and made available to users over the internet. SaaS applications are typically fully-managed by the vendor and accessed through a web browser, with users paying a monthly or yearly subscription fee. In addition to Microsoft Fabric, Cloudera Data Platform (CDP) [7] and Databricks Data Intelligence platform [8] are examples of SaaS based offerings that include integrated data movement, compute and storage capabilities.

IaaS[2]/PaaS[3] deployment model. To make matters more complex, the optimal trade-off between SaaS and IaaS/PaaS depends on the data source.

In a broad sense, the thesis addresses a significant technical challenge - implementing a middle layer that prepares data from on-prem data sources to be ingested into the Fabric cloud system. To achieve this, a deep understanding of the final to-be analytic data management architecture of the EVR is necessary, which further buttresses the arguments for allocating a large proportion of the thesis towards describing technologies and platforms that are strictly not employed at the code level. Furthermore, pragmatically, as the EVR is quite a large company, it would be impossible to design and implement a whole new analytic data management architecture under one study.

The current work contributes significantly by preparing EVR for a Microsoft Fabric-based proof of concept study with Microsoft's Baltics cloud services partner company. The future proof of concept study will involve the same datasets and analytic questions presented in the thesis's two end-to-end practically implemented cases; these include analysing legacy railway level crossing data to identify abnormally short train passing events and analysing and enriching row-level data from incoming purchase e-invoices XMLs. The upstream steps from the thesis' practically implemented end-to-end cases are almost fully re-usable in actual Fabric proof of concept study. Downstream steps from the thesis' practically implemented end-to-end cases that transform and clean analytic data for the reporting layer allow for practically informed comparison and choice between running those data cleaning and transformation operations on-prem/IaaS/PaaS vs utilising Microsoft Fabric. In other words, for the data pipeline downstream steps, the thesis will give a reference implementation case for a fully on-prem solution, and the future Fabric-based proof of concept study with an external partner will give a reference cloud-native solution. Finally, the reporting layer, Streamlit dashboard for level crossing log data and Power BI dashboard for enriched purchase e-invoices data, will be fully re-usable in an actual Fabric proof of concept study and later in day-to-day usage.

In many respects, both practically implemented use cases exceed the complexity of EVR's previously implemented analytic solutions. Practically implemented use cases are based on unordered source data, from which the analytic output can only be obtained through algorithmic data mining (level crossing log data) or from which the source data must first be fetched and enriched across data sources (purchase invoices XML-s). Furthermore, the dockerised and version-controlled deployment model is novel compared to the current EVR analytic solutions. Most importantly, disregarding the analytic output managed by source systems, the current use cases are implemented to make them stateful - the implemented solutions save cleaned analytic data and keep the state information about already included source data. This situation contrasts with the current analytic solution, which relies entirely on the source system to save data permanently. Last but not least, the log data Streamlit-

[2] Infrastructure as a Service (IaaS) is a form of cloud computing that provides virtualized computing resources over the internet. In an IaaS model, a cloud provider hosts the infrastructure components traditionally present in an on-premises data centre, including servers, storage, and networking hardware, as well as the virtualization or hypervisor layer.

[3] Platform as a Service (PaaS) is a cloud computing model that provides customers with a platform, allowing them to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching a data pipeline or app. Example of PaaS based data engineering architecture would be a combination of Amazon EMR Spark serverless compute cluster service [9], Amazon Glue serverless data ingestion and work orchestrator tool [10] and Amazon Redshift fully managed warehouse service [11].

based dashboard is minimal example of reverse-ETL [12] like thinking, by which the cleaned analytic data is feed back to the operational level by allowing users to set per-level crossing alert thresholds which are to be used in future Fabric proof of concept study to send out Microsoft Teams or e-mail alerts about the occurrence of abnormal train passing events, which in turn allows for checking if those alerts were registered by current manual work based abnormal event reporting system.

2.2 Previous Work about Analytic Data Management in the Railway Sector

A brief literature overview reveals that the two main streams of research that deal with big data, data engineering, data warehousing and business intelligence or analytic data management in the railway sector are firstly related to the general topic of asset management and predictive maintenance and secondly to the general topics of railway network and asset topologies, including the aspects of geospatial data modelling and metadata and semantic modelling. Both topics are relevant for EVR as the key strategic goal of the company is to digitise asset management and asset maintenance processes, thus, in the long run, replacing static maintenance schedules with predictive near real-time data-based maintenance scheduling. Secondly, efforts are being made to revamp and re-order the EVR's topological data master data schema, which includes specifying the truth sources and data producers for each main topology category.

Tutchet [13] highlights the importance of ontology-driven data integration in addressing the complexities of railway asset management (including maintenance and monitoring) and the potential benefits of using semantic data modelling for improved decision-making and analysis in the rail industry. This line of thinking means structuring and organising data that emphasises the meaning and relationships between data elements. Unlike traditional data modelling approaches that focus on the structure and format of data, ontological semantic data modelling aims to capture the real-life semantics of data elements and their relationships [13].

Integrated and cleaned data is the bedrock for building business-relevant machine learning tools [2]. In the railway sector, most of these efforts relate to building predictive machine learning models to use as input for maintenance planning. A recent comprehensive survey of the railway sector by Davari and colleagues [14] provides a thorough empirical overview of the current state of data-driven predictive maintenance in the railway industry. Predictive maintenance is a proactive maintenance strategy that uses data analysis, monitoring, and predictive analytics to predict likely equipment failure. The authors categorise the prediction efforts in two broad categories: failure prediction (and its subtask anomaly detection and failure classification), which involves forecasting the approximate time when a failure might occur in a system or equipment; and secondly, remaining useful life estimation which tells how much operational time remains before a railway device needs to be repaired or replaced. Relevant to the current thesis' level crossings' log data mining implementation is Davari and colleagues' [14] conclusion that a big research gap exists in anomaly detection from time-series data.

An older review by Ghofrani and colleagues [15] from 2018 emphasises the central role of predictive maintenance in big data push in the railway sector. Furthermore, the authors bring up examples of big data-based decision support tools being used for more efficient capacity allocation planning and general day-to-day operations management. Review article by Binder and colleagues [16] summarised the findings of 24 relevant scholarly works. Binder and colleagues [16] reach a generally similar conclusion to Davaris and colleagues [14] in their industry-specific field survey, with their unique contribution being the proposing a nomenclature of components, predicted defects, and maintenance conditions targeted by predictive maintenance efforts. They highlight employing machine and deep learning model nomenclature broadly similar to Davari and colleagues [14]. Further relevant to the current thesis, Binder and colleagues argue that the broader societal benefits, mainly safety and resource usage efficiency, increase due to employing predictive maintenance depending on source data quality and data standardisation.

All in all, previous works about data engineering in the railway sector mainly relate to specific problem area and focus on a narrow set of applied methods. The author found no overarching studies that would have tried to describe the railway sector's data engineering journey at a more macro level.

2.3 Key Data Engineering Concepts

The following section will provide a brief overview of the key theoretical concepts. This section is meant to serve as a high-level overview of the process of obtaining and managing analytic data in data engineering. Hopefully, this section will serve as a CliffsNotes-like overview for the later, more technical sections of the thesis, where more in-depth architectural and implementation details are discussed.

The data engineering lifecycle is a structured framework that guides the process of managing and transforming data from its raw form to end state that is ready for consumption by various stakeholders, e.g. machine learning models and business intelligence reports. Following Figure 3 by Reis and Housley [2] encapsulates the key steps and components of the data engineering lifecycle.

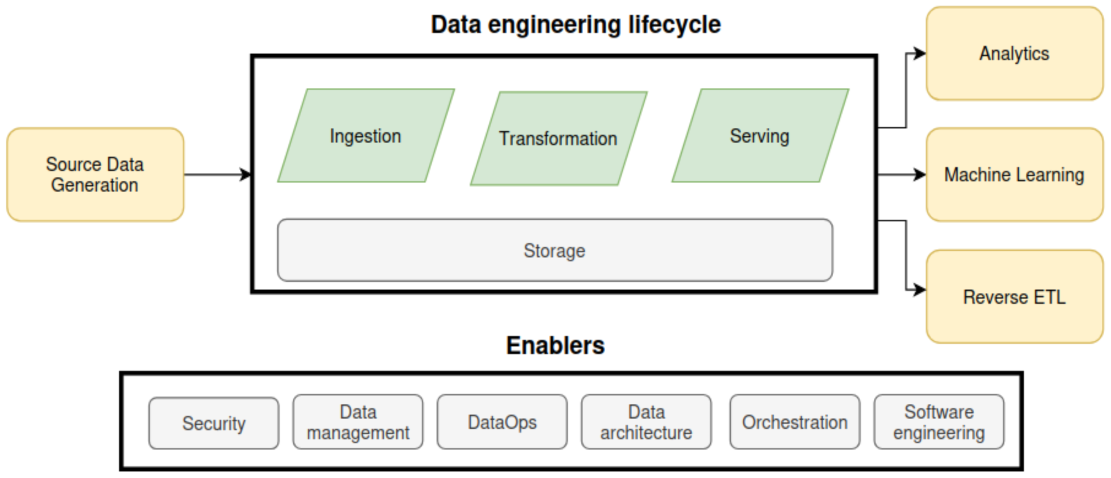


Figure 3. Data engineering lifecycle (adopted from Reis and Housley [2])

Reis and Housley [2] define the key steps of data engineering lifecycle as follows:

1) The first stage of the data engineering lifecycle is the data generation, which involves collecting data from various sources such as databases, IoT devices and sensors, and external API-s.
2) After the data is generated, it needs to be stored. Storage and technical data accessibility is a crucial bedrock on which to the following three steps of data ingestion, transformation and serving depend. This is because, usually the input/output operations towards the permanently stored data are most costly in terms of latency [3, 17].

3) The ingestion stage involves moving data from the storage layer to the processing layer where it can be transformed and cleaned.
4) The transformation stage is where the raw data is processed, cleaned, and transformed into a data model and format that is suitable for analysis and reporting.
5) The serving stage involves making the processed data available to data analytics, data scientists (for machine and deep learning), and other stakeholders through dashboards, reports, and increasingly through API-s. Of importance in the context of the current thesis' log data analysis use case is the concept of reverse ETL. This is an umbrella term for a pattern of data integration where data is moved from a data warehouse, data lake, or reporting systems back to operational systems enabling organisations to bridge the reporting-action gap [18] and leverage insights gained from data analysis or machine learning models in enriching operational systems with the processed data.
6) Ingestion, transformation and serving stages must be orchestrated, meaning the process of coordinating and managing the execution of multiple data processing tasks or jobs to ensure they run reliably in the correct sequence. Often the orchestrator is the singular interface for a data engineer to define, visualise, and manage end-to-end data workflows comprising multiple tasks, transformations, and data movements. Furthermore, the orchestration layer must also handle the complexity related to handling dependencies between tasks, including by automatically resolving dependencies based on task completion status and scheduling the execution of data jobs at specific times or intervals based on predefined triggers (sensors) or conditions.

ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) are common data integration processes of data engineering to move and process data from source systems to landing analytical systems [3]. Following Figure 4 illustrates the difference between ETL and ELT.

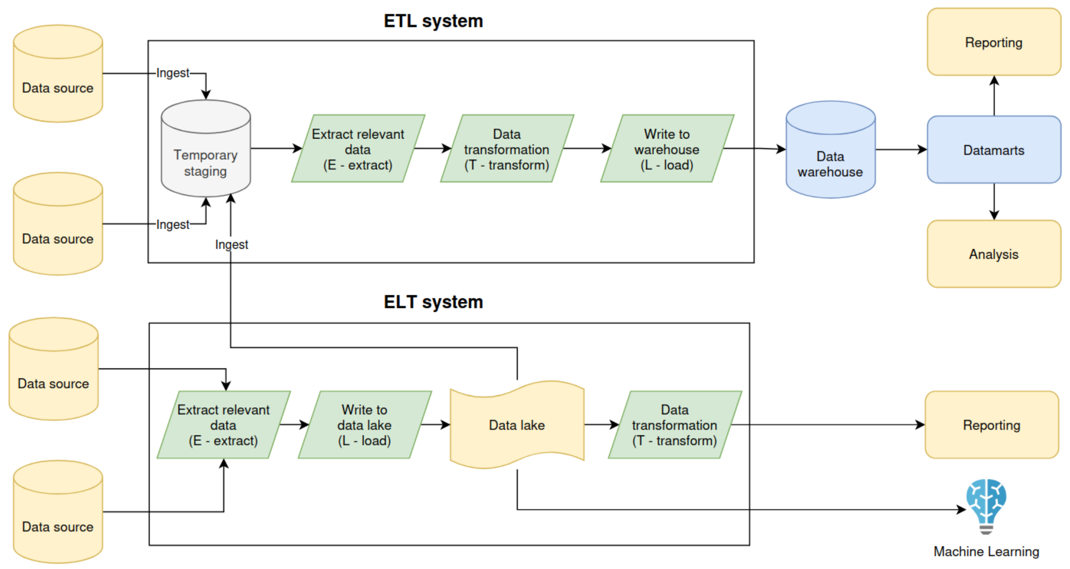

ETL system
Data source
Ingest
Temporary staging
Extract relevant data (E - extract)
Data transformation (T - transform)
Write to warehouse (L - load)
Data warehouse
Reporting
Datamarts
Analysis
Data source
Ingest
Ingest
ELT system
Data source
Extract relevant data (E - extract)
Write to data lake (L - load)
Data lake
Data transformation (T - transform)
Reporting
Data source
Machine Learning

Figure 4. Illustrating example of the ETL and ELT processes (based on [2, 3, 19, 20]) [4]

[4] There are some differences between authors in defining the scope of the staging layer in typical ETL based DW. Staging area can be either a simple periodically updated copy of source database that lessens the load on the source system during ETL process or staging area can be a full fledged ELT-based data lake in itself. To

For the ETL process, data is first extracted from the source system or systems, after which, most importantly, the extracted data is transformed through a combination of cleaning, standardisation, enrichment and aggregation operations to conform to the requirements of the target system's analytic data model. Once the data is transformed, it is loaded into the target data warehouse, data lake, or database for storage and subsequent analysis, reporting, or input for machine learning or deep learning applications.

As was the case for the ETL process, the ELT process begins with data extraction from source systems. Extracted data is in its raw form and loaded into the target storage (usually data lake) or analytical system without transformations. Once the data is loaded into the target storage, transformation processes are applied within the target environment as needed. The previous point encompasses the main difference between ETL and ELT: for idealised cases of ETL, data transformation occurs before loading data into a data warehouse or other target system, while in the case of the ELT, transformation is typically performed after data loading within the target environment. Furthermore, for the further discussion about VJS system's role as data source, it is noteworthy to emphasise that ETL processes data outside the target system, often in a separate ETL engine or source system module, whereas ELT processes data within the target (usually data lake related compute cluster) system. ELT has evolved and is mostly the standard due to low storage cost and the emergence of SQL engines (e.g. Trino, Presto), which together allow for fast DW capable of fast data processing.

The data transformation step is usually not a singular operation; furthermore, often, the stepwise cleaned, ordered and refined data is retained in separate tables. Databricks' Medallion Architecture exemplifies previously outlined logic [21] in which data refinement is described through the logic of stepwise refinement. In detail, the first, i.e., the bronze layer, contains the data in its raw, unprocessed form (without any filtering, cleansing, or transformation). The bronze layer has conceptual similarities with the classical data warehouse's staging area [3]. The next layer of refinement is the silver layer, obtained through data operations like cleaning, transformation, and enrichment. The top, i.e., gold, layer of the Medallion Architecture is where data is ready for reporting and being highly aggregated.

A data lake and a data warehouse are both analytic storage systems used in data engineering for storing and analysing large volumes of data [2]. A data warehouse is a relational database system optimised for querying and analysing structured data to support business intelligence and reporting [2, 3]. Data warehouses use a columnar or row-based storage architecture to store structured data in tables with predefined schemas [2]. Warehouses are typically optimised for query performance and exhibit complex database-world-derived built-in query optimization techniques [2, 3]. Importantly, data warehouses follow a schema-on-write approach, meaning the data must be transformed and structured to comply with the predefined schema to be loadable into the warehouse [2, 3].

A data lake is a centralised repository that allows organisations to store large amounts of raw, unstructured, semi-structured, and structured data at a relatively low cost [2]. On-prem data lakes typically use scalable distributed storage systems, such as Hadoop Distributed File System (HDFS); hybrid and cloud-based data lakes typically use cloud-based object storage services (e.g. Amazon S3 or Azure Data Lake object storage), to store diverse data types, including text, images, logs and sensor data [2, 17]. Data lakes align with the schema-

emphasise the later possibility, the figure presents a somewhat uncommon connection from data lake to ETL's staging area.

on-read approach, meaning that data is stored first, and its structure is applied during analysis or processing [2, 19].

In practice, pure data lakes and data warehouses are uncommon; elements of data lakes are often used for ingestion or staging layers followed by structuring warehouse-like layers [17] (see also Figure 5 for comparison between data warehouse, data lake and data lakehouse). This has led to the development of a relatively new architecture called data lakehouse, a data management system that combines on a single platform the features of a data lake and a data warehouse [19, 17]. Data lakehouse aims to provide the scalability and flexibility of a data lake with the management and performance capabilities of a data warehouse [17, 19]. At the technical level, the data lakehouse stores raw data like a data lake (mainly in the format of Apache Parquet or ORC (Optimised Row Columnar)) but also supports ACID[5] compliant transactional data management features, data versioning, managed schema evolution and indexing like data warehouse [17, 19] through its metadata management, indexing and caching layer [17] provided by technologies like Delta Lake [17, 19]. In detail, Delta Lake maintains a transaction log that records all the changes made to the data lake. This transaction log is stored in a Parquet format within the data lake [17, 19]]. Delta Lake stores metadata about tables, partitions, and data files in the transaction log, which enables Delta Lake to track the lineage of data changes and manage schema evolution [17, 19]. Delta Lake is tightly integrated with Apache Spark, meaning that users can leverage Spark's distributed processing capabilities to interact with Delta Lake tables to perform complex analytics tasks encompassing large datasets [17]. Apache Spark is a general-purpose single-node and cluster computing system that provides high-level APIs in Java, Scala, Python, and R that is able to handle data in batches and real-time streaming [22].

[5] ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. Those properties are needed to ensure reliable data transactions with the database system. Atomicity ensures that a transaction is treated as an atomic unit of work that either succeeds totally or is fully rolled back. Consistency ensures that a transaction move the database from one valid state to another valid state. Isolation means that concurrent transactions can be carried out without interfering with the parallel transactions. Durability means that state changes created by transactions are durably saved. [2]

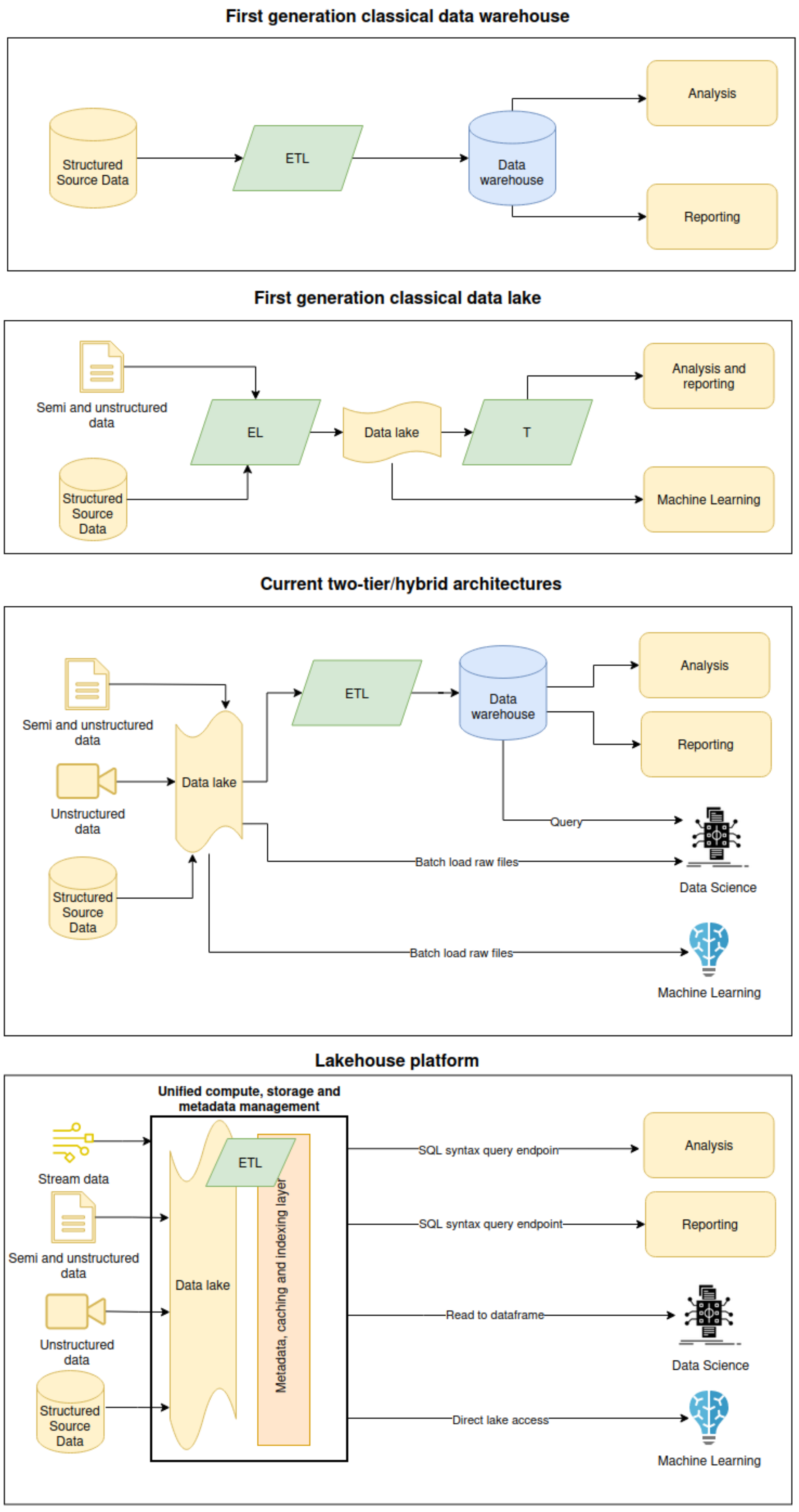


Figure 5. Idealised architectures of data warehouse, data lake, combined two-tier architecture and data lakehouse (based on [2, 6, 8, 17, 19])

It is crucial to note that data in the live system's transactional tables and data extracted and saved in the data warehouse or data lakehouse is quite different. Simitsis and colleagues [3] highlight the following key differences between analytic and transactional live system data. Information system-level data, called OLTP (Online Transactional Processing), is designed and modelled for transactional processing to support daily live operations. In detail, the OLTP data is typically normalised (preferably in the III standard form), and the general data model is optimised for low latency, i.e. fast individual transaction processing, with the majority of the workloads being CRUD (Create, Read, Update, Delete) operations on individual records. Therefore, mainly, the row-store is used. While the data size is lower (in gigabytes to terabytes), OLTP systems place high demand on concurrent operations, i.e. the need to serve multiple user sessions simultaneously.

In contrast, OLAP (Online Analytical Processing) data models are designed for analytical processing to support complex queries and data analysis. The analytic data is modelled and saved to optimise a few concurrent but long-running complex query performances and aggregations. The analytic data is often denormalized or saved in the star/snowflake schema to optimise query performance; e.g., the popular reporting tool PowerBI assumes that the input data is presented in the star schema.

Analytic data modelling and storage based on star or snowflake schemas follow the general principles of dimensional modelling. Kimball and Ross [29] emphasise the following key concepts in analytic data modelling. Fact tables contain quantitative data, also known as measures, representing the business metrics or atomic meaningful business events (e.g. singular sale, accounting entry etc) that are being analysed. Fact tables are modelled through dimensional tables, which provide context to the data stored in the fact tables and ensure efficient filtering, grouping, and aggregation of the data in the fact table. The relationship between fact tables and dimension tables is established through foreign keys. Dimension tables contain attributes that are represented as descriptive columns that provide additional information about the data. Crucially for drill down and rollup query performance, dimensions can be presented in a way that models intra-dimensional hierarchical relationships. Relatedly, often in the context of modelling hierarchies, a star schema is the most straightforward way to do dimensional modelling, just surrounding fact tables by denormalized dimension tables (e.g. singular dimension table for department with division grouping and crucially division's attributes being represented as separate columns). In a snowflake schema, dimension tables are normalised by breaking down hierarchies or attributes into separate foreign key-linked tables (e.g. two foreign key-linked dimension tables: one for a department and its attributes and the other for division and its attributes with a department table also containing foreign key reference to division table id field).

While Kimball's model is built from the bottom up, its alternative, Inmon's data model, takes a centralised, top-down approach in which data is integrated from across the organisation into a granular, highly normalised ER model [2]. In extreme cases, the data in Inmon's data warehouse, especially if it is entirely in third normalised form, closely resembles the normalisation structure of the source system itself [2]. While Kimball and Inmon emphasise the organisation of business logic within the data warehouse, the Data Vault proposes a different data modelling method. The Data Vault model comprises three primary table types: hubs, links, and satellites. In essence, a hub stores business keys, a link preserves relationships among business keys, and a satellite embodies the attributes and context of a business key. Users can retrieve the information they need when they query a hub that links to a satellite table containing the applicable attributes [2]. In industry practice, Data Vault and its derivative Data Vault 2.0 are more resilient in handling source data schema drifts [23].

Interestingly, according to Tutchet's article touched briefly in the previous section [13], already ten years ago, the increasing volume and heterogeneity of data in railway systems had already posed challenges for traditional analytic data management techniques like data warehousing. Decade-old thinking by Tuchet aligns very well with modern domain-based thinking from designing microservices [24, 25] and data mesh-based analytic data management [26, 27]. Data mesh is an analytic data management architecture emphasising decentralisation and domain-oriented data ownership. It involves organising data into domains managed by individual teams within an organisation, with each team responsible for understanding and producing data specific to their domain. The fundamental concepts of data mesh include treating data as a product, implementing self-serve data platforms for teams to manage their domains autonomously and utilising a federated approach to data governance. As a sneak peek, the thesis employs data mesh in the proposed architecture to organise cleaned analytic data. Furthermore, as the data mesh data management model is as much a governance or data leadership model as it is technical architecture [26], a notable proportion of the thesis is allocated for thinking through the nexus between technology choices and their assumed governance and usage by the broader end-user groups of the organisation.

In the upcoming sections, the thesis will uncover details of multifaceted levels of aggregation within the EVR source data systems. Hence, it is essential to carefully consider how OLTP tables are logically translated into analytic domain objects. In the field of data engineering, a domain object refers to a data model or structure that represents a specific and meaningful concept within the target domain. These objects are often created to encapsulate data and behaviours that are relevant to the addressed problem domain [24]. Domain objects are a critical aspect of an application's architecture's business layer, aimed at reflecting the real-world entities and relationships within a particular domain or area of interest. In the context of data engineering, domain objects play a vital role in ensuring that the design and implementation of architectures align with the real-world entities and relationships they represent. This alignment facilitates encapsulation of complexity [28] and helps in achieving intuitive data models, transparent translation of business logic to the code, and ultimately, systems that are more maintainable and adaptable to the evolving needs of the business [24, 26].

At the technical level, it means striving towards coherent stateful entities with well-defined business meanings. The crucial concept for analytic data management and engineering is domain aggregates, which means aggregates based on sources stored across many OLTP tables in source transactional systems. In the domain-driven design (DDD) system architecture paradigm, domain aggregate is a collection of entities and value objects that logically, i.e. in terms of the domain's business rules, belong together and are always consistent concerning invariants[6] [24, 25, 28]. Further, in the DDD paradigm, writing ad-hoc database queries to modify domain objects belonging to domain aggregate is an anti-pattern [24, 25, 28]. Instead, the repository pattern is enforced to encapsulate storage access code and provide a straightforward interface to retrieve, store and remove domain objects from the domain aggregate [24, 25, 28].

The take-home message from the previous theoretical discussion to the current thesis' TO-BE architecture is that in the ideal case, the analytic/data-warehousing layer should contain something other than the code that reconstructs domain objects and aggregates from the system's raw transactional tables. Instead, it is a non-functional requirement for source

[6] I.e. the whole state of domain aggregate is updated coherently if there is change in element of the collection comprising the domain object.

systems to be apt to provide an analytics/data warehousing layer with a coherent set of business domain model-following objects and aggregates (e.g. via API endpoint). In that sense, the source system's data is describable as a product (valuable, accessible, documented) [26] or open host service [24, 25], meaning that the source system, for example, adapts its API endpoints to the needs of the accessing parties.

3 Outline of AS-IS State of Analytic Data Management and Business Intelligence in EVR

In this chapter, the thesis will delve into EVR's current technical setup for managing analytic data and data sources. While it may seem excessive to describe source data systems like PONY, VJS, TTCSM, and SharePoint - which are not utilised in the practical implementation use cases of chapter five - it is necessary to paint a complete picture of EVR's analytic data management and business intelligence journey. EVR has already made significant strides in reporting and central and system-level analytic data management. Therefore, proposing a new TO-BE architecture without considering the current AS-IS state would be unwise and inefficient. Thus, this chapter will outline both the strengths and areas for development in the current state. It's important to note that any shortcomings in the current state should not be taken as a harsh critique. EVR has already achieved a respectable level of maturity in analytic data management, having already picked the low-hanging fruit. To continue advancing in data engineering, EVR must focus on building the capacities outlined in this chapter's development needs.

3.1 Description of the Current Technical Setup

In EVR, Microsoft Power BI is the current go-to tool for business intelligence needs. A trusted third-party collaborator is responsible for generating new Power BI reports in response to requests from the business side or upon the availability of new, report-ready datasets. It is worth noting that this partner manages Power BI service deployment settings on behalf of EVR. The partner has prepared over 25 reports conveniently accessible to around 100 end-users via the Power BI online service. These reports exhibit a high level of user experience and design.

Technically, Azure VM running MS Windows Server, generally denoted as EVR's data warehouse (EVR DW), is the data source for relational data for the Power BI reporting service. Architecturally, the MS Windows Server VM runs an MS SQL Server instance, including the orchestration component of SQL Server Integration Services (SSIS) accessible via MS Visual Studio GUI and Power BI Data Gateway service installation. The cloud VM is deployed inside Azure's vNet infrastructure and connects with EVR's on-premise systems via the VPN-vNet gateway [30]. As for more straightforward tabular cloud-hosted data sources, like MS Sharepoint, the current solution ingests data through Power BI-s built-in data connections. Figure 6 describes the current technical setup of EVR's data warehouse.

Before describing the data sources and data extraction, transformation and enrichment steps employed in the EVR's DW in more detail, it is essential to note that the paramount architectural decision of the current setup is making the EVR's DW non-stateful. MS SQL server instance of the DW does not permanently store any data. Relational data sources are extracted as linked tables, linked source system's materialised views or, as an often-used backup solution, as copies of source tables. In the latter case, the SSIS package orchestration job truncates respective landing tables in EVR DW's MS SQL instance before the update. After which, the orchestration job carries on by non-incrementally loading complete, up-to-date copies from the source systems into DW's MS SQL tables. With few exceptions, the relational data in DW is updated once a day, at nighttime, in order to minimise the I/O load on source systems. For reporting layer output, EVR's DW presents relevant MS SQL tables for Power BI semantic model generation as views through the on-prem data gateway.

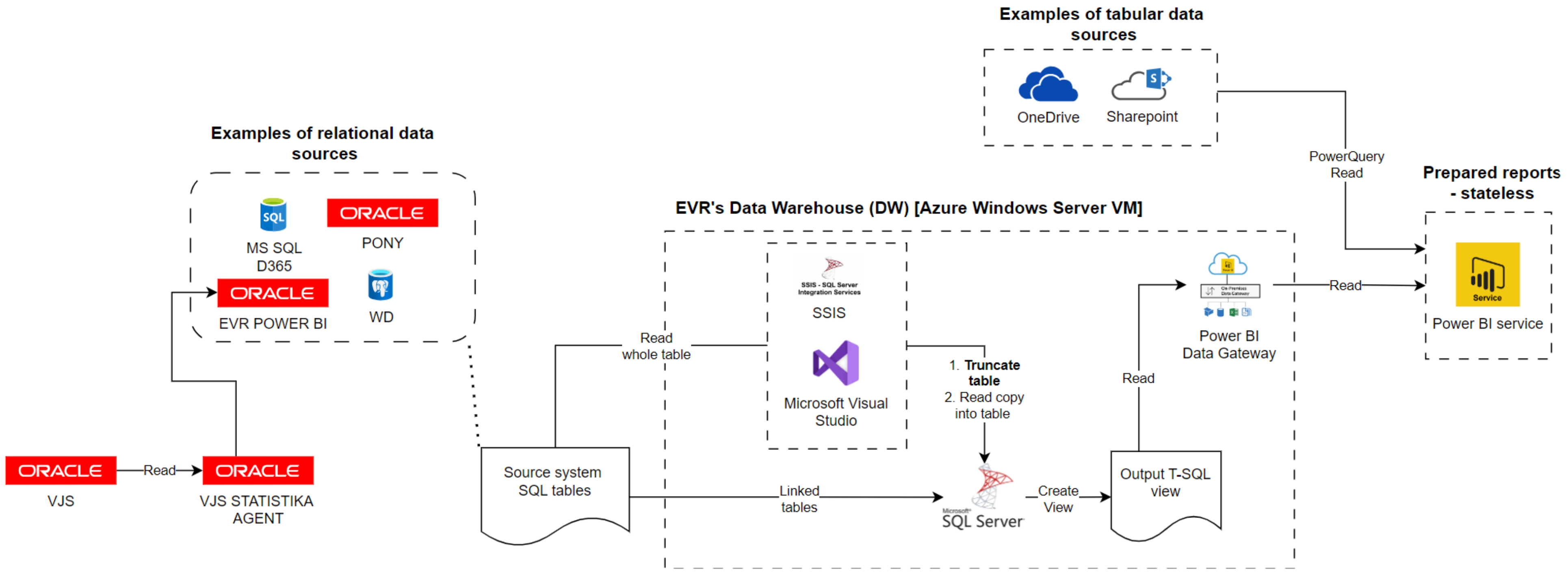


Figure 6. Outline of the AS-IS analytic data management architecture.

3.2 Analytic Data Landscape of EVR

Data transformations based on relational data have differing complexity and locus of data cleaning and transformation location, i.e. between EVR's DW vs source system. The most complex data transformations relate to EVR's wagon management system (VJS), which also presents the extreme of relying on the source system to extract, clean and transform the analytic data. VJS is a monolithic system based on the Oracle database platform, in which the database platform manages all the data storage and nearly all the programmed business logic (PL/SQL procedures). To illustrate VJS's complexity, the VJS's codebase consists of over 2.3 million rows of SQL DDLs and PL/SQL procedures.

VJS's schema STATISTIKA_AGENT encompasses a set of extensive partially parameterized PL/SQL procedures that transform the system's OLTP live tables to clean aggregated analytic data tables. By and large, the data pipelines implemented in the analysis schema first extract complete business-relevant events (e.g., a train journey from station A to B) from atomic events described in OLTP tables and tie those business-relevant events to concrete business objects (e.g. train) and enrich business event-object combinations with relevant characteristics (e.g. nomenclature and mass of goods carried). As source transactional tables are in constant flux, extracted and cleaned business-logic-focused analytic input tables are timestamped and versioned. A further set of stored PL/SQL procedures creates a dimensionalised, cleaned and highly aggregated analytic output, realised as tables and materialised views. As a security and compatibility compromise with DW, the output set of tables and materialised views are redundantly copied to the newer on-prem instance of Oracle DB, which the EVR's DW accesses (see also Figure 6). Of note is that a group of EVR personnel uses locally stored and manually executed SQL queries to get analytic output that is not reflected in DW-based Power BI reports.

Source data from the PONY[7] multidomain information system illustrates a more reasonable compromise between data transformations in DW and the source system (see Figure 6 for reference). Similarly to VJS, PONY runs on the Oracle DB technological stack. However, contrary to VJS, PONY's relational data model is almost entirely in the third normal form, with all the main business-relevant entities presented in separate tables and those tables covered by well-defined dimensional tables that include dimensions' metadata and validity status. As a corollary, operational data in PONY loosely resembles data the reporting layer would access from traditional, i.e. stateful, DW that follows the Inmon data warehouse data model. Therefore, even the current EVR's DW can create star schema-following Power BI semantic data models using straightforward join-based transformations on a limited number of linked tables from PONY.

Regarding custom code amount and complexity, EVR DW's most complex transformations concern the MS Dynamics 365 ERP system, namely ERP's MS SQL database. As a complexity highlight, the developer has implemented user-updatable rudimentary parametrization for project budgeting reporting, which uses an end-user-updatable MS Sharepoint table that serves as an input in the respective SSIS job package. On the contrary, relatively more uncomplicated source data originates from tables and files embedded in Sharepoint sites (formerly also from Excel files stored in OneDrive). In the latter case, usually, one SP site serves one Power BI report.

[7] PONY is EVR's internal information system. It is built over time to support a diverse and loosely coupled set of business functions (incident management, hazard assessment and management, management of infrastructure master etc).

The current EVR's DW does not analyse the technical system's log data. Most legacy control systems have non-configurabele monitoring environments with end-user-facing GUI. Suppose one needs more detailed custom analyses that are not covered by an out-of-the-box interface. In that case, the end user extracts the data as an Excel file and analyses it locally on their personal work computer.

To conclude the high level and overview of the data landscape with a forward-looking note, the data landscape in EVRs will get more complex. Multiple new technical systems (e.g., CCS systems by Siemens and MIPRO) will produce more varied real-time log data. This data needs to be cleaned and centrally stored. Real-time monitoring data is planned to be used to implement a predictable maintenance scheduling of assets instead of using old, partially from Soviet times, static maintenance schedules. Moving from static to predictive maintenance scheduling is also one of the company's strategic goals.

3.3 Strengths of the Current Setup

The following section gives an overview of the strengths of the current setup, with a focus on aspects that should be preserved and carried over to the TO-BE architecture.

The VJS's analysis module (i.e. STATISTIKA_AGENT schema) has a long history and a large codebase. Therefore, it contains an operationalized description of protocols for translating live transactional source data into analytical business-relevant domain objects and aggregates. Contrariwise, analytic protocols and principles have yet to be documented, and the primary source of truth is the PL/SQL procedures themselves. Furthermore, solutions for different corner cases have been incorporated into the code for a long time. Therefore, the VJS in-built analysis module acts as an anticorruption layer [24] between the live system and reporting, e.g. providing computation paths in case the same but conflicting live data is present in different VJS's OLTP tables. Furthermore, persistent data storage and basic versioning of the data are features of VJS’s analysis module that are not present in any other EVR's custom business intelligence and analysis solutions.

Notably, in the current data landscape, the VJS analysis schema and related tables and procedures are the most evident implemented examples of employing the principle of domain modelling (see theoretical background chapter for more details). A future case in point is the architecture of EVR's new in-development traffic management system, TTCMS, which follows the pattern that transactional OLTP tables are closed-source at the database level for the deployer of the system. Instead, the TTCMS information system provides a coherent set of domain objects and aggregates in a database schema accessible to the analytics and data warehousing layers. A further future case in point is the evolution of Microsoft's Dynamics 365 ERP suite (D365), which in its cloud version restricts clients' direct access to the underlying MS SQL database that is available in the legacy on-prem version of the ERP suite. EVR will soon migrate from the on-prem D365 installation to the cloud-based version.

From the code perspective, the procedures in the VJS STATISTIKA_AGENT schema strongly rely on stored PL/SQL procedures from other operational schemas. Operational procedures are loosely used as API endpoints that at least partially mask the operational data logical-level complexity from the analysis schema-specific procedures. Corollary to the current program setup, the load and dependence on the source system would remain high even if the logic inherent in the STATISTIKA_AGENT schema were migrated outside of VJS. Furthermore, narrowly refactoring and migrating only the analysis-specific procedures to a new platform would require substantial investment, which would not immediately add new functionality or business value. To bolster the argument for keeping the current VJS's analysis schema intact, translating data transformation logic performantly from Oracle's

PL/SQL, let us say to Spark[8], would be a substantial undertaking, as PL/SQL has constructed operations that do not have clear one-on-one correspondence from Spark. Likewise, writing Spark code that relies highly on external Oracle's stored procedures would not be trivial.

Last but not least, the current portfolio of Power BI reports is the strong point of the EVR's data landscape. Reports are visually pleasing, modern-looking, logical to use, and generally fast, with few exceptions. Power BI as a platform has proven itself in the eyes of EVR's end users.

3.4 Development Needs for the Current Setup

This section gives an overview of the deficiencies of the current setup, with a focus on aspects that should be solved, or at least highly improved, by implementing TO-BE architecture. Areas that need improvement are presented followingly in tabular form. First, the specific problem is defined (see column Problem description). Secondly, for every problem description, the author of the thesis highlights the ability gap underlying the problem. Therefore, the thesis describes deficiencies in current tooling and general processes that make the problem unsolvable with capabilities available in the AS-IS state.

Table 1. Development needs for the current setup.

Problem definition (PRD)	Ability gap (AG)
PRD1: Governance, documentation and finding business value of data assets.	AG1; Metadata management system[9]. AG2: Promoting data ownership and self-serve culture (data culture) [24, 26]. AG3. Enforcing business-side data owner role.
PRD2: Documentation of currently implemented business intelligence solutions needs improvement.	AG4: Documentation standards (e.g. docstring system). AG1: Metadata management system. AG5: Clear analytic data pine programming guidelines (tools and project structure). AG6: DataOps practices: a) Code repository and versioning usage in creating

[8] Apache Spark is characterised as a processing framework with a very complex API that relies heavily on code, requiring the code writer to take on the responsibility of optimization tasks that are typically automated in SQL-based engines. Therefore, when utilising Spark, data engineering teams must proactively address the challenges associated with Spark optimization, particularly for resource-intensive and prolonged jobs. [2]

[9] Metadata management is a framework that encompasses processes, policies, and technologies for cataloguing information assets within an organisation, ensuring their accuracy, integrity, and usability. It serves as a foundational practice for data discovery, search, collaboration, quality, and governance, employing various tools (often SaaS products) to manage all metadata types. Metadata is categorised into passive metadata, which includes basic data definitions like schema and data types, and active metadata, which adds context by detailing all interactions with the data. [31]

	analytic data pipelines; b) CI/CD[10] practices, including infrastructure as a code and Dockerised software deployment to improve integration between different technologies; c) swift data onboarding (i.e. the usage of standardized data ingestion tools).
PRD3: The current EVR's DW lacks some defining features of classical DW.	AG7: Single place to persistently store at least gold layer analytic data.
PRD4: Currently implemented analytic solutions are fragmented and lack overarching data architecture.	AG7: Set of architectural principles for EVR’s analytic data management. AG8: Agreeing and enforcing analytic data modelling principles, at least for the gold layer of the analytic data (e.g. Data Vault). AG1: (Active) metadata management system.
PRD5: The current technological stack is complicated to extend and partially technologically amortised.	AG9: DataOps[11] practices. AG10: Observability tools[12].
PRD6: Currently implemented analytic solutions are monolithic and hard to maintain.	AG9: DataOps practices. AG10: Observability tools.
PRD 7: Currently implemented analytic solutions are challenging to reuse/refactor.	AG9: DataOps practices. AG11: Agreed upon architectural principles for analytic data related developments. See also AG8.

In terms of *governance, documentation and finding business value of data assets* (PRD1) EVR's data landscape has yet to be thoroughly mapped and documented. Currently, many critical large information systems need more quality documentation. Even if certain data assets are well documented, the discoverability of this documentation could be improved. Relatedly, the ease of improving the documentation is a challenge as there is no up-to-date master catalogue of data assets that is a) accessible to a broader audience, b) easy to use, c) searchable and d) reflective of the actual state of the live systems.

The current analytic solutions focus on solving the reporting domain, including automating repetitive data analysis tasks and increasing the observability of the data quality in source

[10] CI/CD, short for Continuous Integration/Continuous Deployment, automates merging code changes and deploying them to production, thus facilitating rapid, reliable software development and deployment cycles through enhanced collaboration and accelerates delivery.

[11] DataOps is a discipline that focuses on streamlining data and analytics processes through automation, collaboration, and continuous improvement. It combines principles from agile software development and DevOps to enhance the speed, quality, and efficiency of data operations within organisations [32].

[12] Observability in data engineering refers to fully understanding the state of the data engineering systems. This means tracking and monitoring data flows, infrastructure, applications, and systems in real-time to identify issues, tune performance, and ensure the reliability and integrity of data processing [2].

systems. Reports are created following the waterfall or supply push pattern, which centrally develops reports with a focus on the end-user-facing UI of the solution.

Although successful, the current pattern of developing analysis solutions might need to be more integrated and broader. The current focus on readymade reports as a product instead of the underlying data as a product leads to an overly substantial client-supplier cooperation pattern between IT and business parties. This asymmetrical cooperation pattern diverts the focus from the business side's responsibility, technically, of course, in cooperation with the DW/data engineering team, of organising their domain information in a documented, discoverable, and integrative/reusable way. Furthermore, the previously described cooperation dynamic is an attenuating factor in enforcing the broader data owner role in the organisation.

Creating and enforcing the data owner role is paramount, as the simple set of analytic challenges solvable by creating isolated Power BI reports that depend on a singular or sparse number of input data tables that are often pre-cleaned and pre-aggregated has peaked. Complex and not fully predefined fuzzy datasets assume deep domain knowledge and responsibility, i.e. taking the data owner role in conjunction with continuous IT-business teamwork.

Datafication and digitalization mean business rules and actual work processes are increasingly codified and enforced in data structures and application code. In this context, an asymmetric client-service provider cooperation model between business and IT instead of continuous cross-functional teamwork would create a risk of responsibility drift: technical developers and maintainer of the business-critical codebase and data would implicitly assume the role of the grantor of overall coherence of different business processes, i.e. business architecture.

The situation by which *the documentation of currently implemented business intelligence solutions needs improvement* (PRD2) is most clearly seen by the fact that the code running in EVR's DW has not been documented, nor is it committed to any code repository. Relatedly, developments in EVR's DW are not version-controlled. An overview of currently live data pipelines and their statuses is available at the level provided by the Power BI online service by default. Configuration choices and applied settings on all levels of EVR's DW have yet to be documented, and management of those settings has been almost totally outsourced to an outside partner.

As a non-exhaustible list of development needs, there is work to do in understanding how the Azure network serving the VM on which the EVR's DW is deployed has been configured (especially how on-prem and cloud resources network are integrated), which configured SSIS packages serve, which reports, what the applied data transformations are, and partially also what the used data sources for every pipeline are.

Although most previously posited areas needing further documentation are reverse-engineerable from the code or UI, this task still requires substantial work time and yields little immediate business value. The current lack of documentation entails the following corollaries:

1) The situation increases the vendor lock-in risk towards the current business intelligence development partner, as the new partner would need to endure substantial sunken costs in unpacking the currently implemented setup.
2) The reusability of already developed data pipelines and cleaned data sources, as well as the implementation of more complex data pipelines based on the current EVR's DW setup, needs to be improved.
3) Migration to a more up-to-date DW or data lake platform and already implemented code reuse is made more complicated (and costly).

4) A loose and non-documented setup and deployment configuration is a security risk.

Of note, the currently outlined critique does not mean that the code in EVR's current DW is defective or of low quality.

The situation with documentation is similar for analytic data flows, in which the source systems perform data extraction, cleaning and aggregation steps. For example, the complex and monolithic codebase of VJS's STATISTIKA_AGENT schema needs to be more thoroughly documented. Furthermore, due to time pressure and high workload, the corner cases are often solved by ad-hoc code modifications, mostly left undocumented.

Further area in need of development is that *the current EVR's DW lacks some defining features of classical DW* (PRD3). The cleaned analytic data is versioned and permanently saved only in VJS's analysis schema. EVR's DW re-computes data extractions and, if needed, transformations at least nightly across the whole history of input datasets. This architectural pattern might become a bottleneck in future data analysis scenarios where DW must meet the requirement to be able to run computationally more costly extractions and transformations. Iterative data mining based on log data is an example of such a computationally expensive scenario. Similarly, the current EVR's DW solution has yet to provide a readily usable way to manage datasets and metadata, e.g. information about business-relevant keys and master data. Making current EVR's DW stateful, i.e., able to permanently store data, would require re-engineering already implemented data pipelines (SSIS packages) and setting up the scheduled backup of the MS SQL instance.

Concerningly, *currently implemented analytic solutions are fragmented and lack overarching data architecture* (PRD4). As the main output of EVR's DW is the Power BI-based reporting, and the data is not permanently stored, the general data model of warehousing data (e.g. choice between Inmon, Data Vault 2.0 or Kimball) has yet to be agreed upon and implemented.

Notably, there needs to be more overall observability in analytic data flows across its complete cycle from source data to reporting. This problem is especially acute for analytic data flows where transformations occur across different systems, which mostly impedes debugging. For example, EVR's DW uses highly pre-aggregated VJS data, but EVR's DW developers do not have direct access to VJS's analysis schema. Detailed logging information, in conjunction with the observability of data pipelines, is accessible only through the database-specific interface. In the case of data pipelines that combine data aggregation operations by the source system and EVR's DW, there is no single source for logging and observability.

As noted earlier, the locus of data transformations and methods differ depending on the data source. Of note is the current overuse of Power BI's low-code data transformation utility PowerQuery, which might hinder the future reusability and refactorability of already created high-quality reports. PowerQuery is most suited for reports that are created on a self-serve basis. Centralised reports, created as data products, would benefit from having DW engineers prepare the complete domain-specific ready-to-use datasets, as is the case for the datamart DW design pattern, which does not require extensive transformations on behalf of the reporting tool. Data transformations carried out in Python or SQL code are more maintainable in the long term than automatic M code generated by Power BI's drag-and-drop/point-and-click interface. To buttress the argument against PowerQuery, Power BI can almost entirely automatically infer suitable data models, including relationships between tables, through its direct query interface.

The current technological stack is complicated to extend and partially technologically amortised (PRD5). Monolithic relational database engines (MS SQL and Oracle DB) and

Power BI's PowerQuery run most of the analytic data transformations. For VJS, Oracle scheduled stored procedures, and for EVR's DW, SSIS, now often douted as legacy technology, orchestrate data transformation operations. Sharepoint data is mostly transformed using Power BI's PowerQuery.

The current technological stack is most apt for relational and tabular data. Extending the current setup to ingest object-oriented or API-sourced data would take much more work than would be the case when using more modern infrastructure. Similarly, combining different types of data sources in the same data pipeline is more challenging.

Furthermore, extending the current setup with third-party data analysis packages would be complex. Therefore, the current setup is more restrictive than modern data transformation environments, such as Databricks notebooks, which allow more flexibility in configuring and versioning the runtime environment (including installed packages) and combining SQL-syntax-based data transformations and custom Python analysis packages. The current level of reporting deals with arithmetic averages, sums and other group-based aggregates, for which the current setup is sufficient. However, extending the current scope with simple statistical analysis (e.g. simple regression model) or machine learning (e.g. random-forest-based anomaly detection model) would be complicated.

For future scalability, it is essential to note that both MS SQL and Oracle DB instances run on single-node deployment instances.

Generally it seems that *the currently implemented analytic solutions are monolithic and hard to maintain* (PRD6). VJS's analytic schema contains multiple procedures that are several thousand to more than nine thousand lines long, stored as a single file. The code is procedural and could be further decomposed by the concerns and parameterized, which would improve the maintainability of the code. The versioning of the data transformation algorithms has been realised in the code through version-specific conditional branching. Therefore, large proportions of the code are left dormant if the latest version of the algorithm runs. Furthermore, there is no straightforward way to concurrently deploy multiple versions of the same data pipeline. Debugging of the code is complex, as the code cannot be isolated and executed on the local machine, as with more modern DataOps-focused data engineering tools (e.g. Dagster, containerized deployments).

Relatedly to PRD1-PRD6, *the currently implemented analytic solutions are challenging to reuse/refactor* (PRD7). The previously outlined development needs of the current setup in concert lead to the situation in which many of the current setup's main building blocks are challenging to reuse in the future, as well as more modern EVR's analytic data management platform. The interlude is especially noticeable in steps dealing with data transformations and storage. The current setup has been developed piecemeal without pre-specifying the overarching analytic data management architecture. In this respect, the EVR's current DW is more akin to the interface machine, which ensures data access to the Power BI reporting layer.

4 Outline of TO-BE State of Analytic Data Management and Business Intelligence in EVR

In this chapter, the thesis delves deeper into the theoretical concepts discussed in the second chapter. The following chapter expands on the previously described central concepts of data engineering and brings them to life with specific, real-world technology examples and alternatives. This analysis is organised through the lens of decision dimensions for the TO-BE architecture.

The chapter culminates with the presentation of the TO-BE architecture of EVR's analytic data management. Here, the thesis takes stock of the detailed description of the EVR's current analytic data management system and data sources presented in the previous chapter. The TO-BE architecture argues for a flexible hybrid between on-prem/IasS and SaaS solutions with the exact balance between on-prem/IasS and SaaS, depending on the type and complexity of source data.

4.1 Main Decision Dimensions and Technology Alternatives for Future Architecture

The following section of the work gives an overview of the categories of choices that influence the future of EVR's analytic data architecture. The thesis outlines the categories to create an organising frame for selecting critical technologies for the final TO-BE architecture from the overwhelming variety of technologies available, both open source and commercial. The work defines the extremes for each decision dimension with the corresponding technology candidate examples. In most cases, there is no categorical best choice for the individual decision latitude category. Instead, the rationale of this categorization exercise is to exemplify trade-offs between technologies, knowledge of which would help create TO-BE architecture that is more coherent across its components and unavoidable trade-offs.

Decision dimension 1: *Amount of raw data moved between live systems and TO-BE analytic data management platform*[13]. The first extreme case would be ingesting and permanently storing/mirroring almost all the relational tables from the live systems to the future data warehouse/data lake/data lakehouse staging area. As an upshot for this architectural choice, the future data platform should handle all the domain object derivation logic from raw source systems' tables to emanate analytic quality tables that are well documented, business-domain oriented, integrated across sources, non-volatile and timestamped. As an additional functionality, such a setup would act as an incremental backup for the live system in case of batch-based data ingestion or live backup if the future platform uses change data capture (CDC). The simplest way to realise the previously described setup would be a classical DW realised on the row or column/row combined DB engine, such as Oracle DB [35] or MS SQL platform [36, 37]

The other extreme of the data movement decision dimension would be the usage of data federation. In the later case, the centralised query engine would interface structured, primarily relational, data across live systems, ingest the data via queries while keeping it in the process memory, transform the data and load the resulting warehousing quality extracted

[13] The first dimension of decision-making is becoming increasingly uncertain in the frontier of data engineering technologies. Apache Doris is an open-source technology that provides federation and ETL data warehouse/ELT data lake-like capabilities on a single unified platform [33]. In the world of MySQL, TitaniumDB is a neo-SQL technology that offers OLTP and semi-automatically derived OLAP capabilities on the same database platform [34]. This means the TitaniumDB platform can perform analytic workloads and serve as the backend data tier for production systems.

and cleaned analytic data to the permanent analytic storage. With this technology, the locus of analytic domain object derivation from transactional live systems' data is open; it can be accomplishable at the federation level or by the live system. Examples of open-source technologies that can be used to realise federation-based architecture are Trino [38, 39], Presto [40], and DuckDB [41].

Decision dimension 2: *Architecture of schema-on-write or schema-on-read*. This decision dimension also describes a dichotomy between the ETL and ELT pattern of building an analytic data management system in its idealised form.

To recap, schema-on-write extreme is an approach that assumes the ETL procedural model in building data pipelines. In this approach, the target data schema for data loading is predefined, and transformation operations based on the loaded data must create an output that complies with the target data schema. This approach is commonly used to build classical data warehouses and is best suited for stable and structured data sources. However, handling schema drifts can become quite complex, especially if the data warehouse is technically realised on a row-based relational database [20, 35, 42]. If the data size is large and the analytics demand real-time dimensionalised cross-aggregations, columnar OLAP-optimised databases should be preferred. Examples of the technologies suitable for the former are open-source technologies such as Apache Druid [43] and Apache Pinot [44] or commercial offerings such as Clickhouse [45].

The process of schema-on-read assumes an ELT procedural model for building data pipelines, which is the guiding principle for constructing data lakes and also effectively used on conventional and cloud based data warehouses as a basis for building scalable staging layers (see chapter two for details). In schema-on-read, predefined schema compliance is not required during data loading.

State-of-the-art technologies and design patterns, grouped under the umbrella term data lakehouse, aim to enhance the ELT approach by incorporating the strengths of the ETL model. In a data lakehouse, much like ELT, all kinds of raw data, including both structured and unstructured data, are stored centrally in object storage (for example, Azure blob storage and Amazon S3 from commercial offerings or open-source S3-compliant option Minio S3). The data is then transformed and structured using high-speed parallelized computing engines such as Apache Spark [22], DASK [46] or DuckDB [41], Trino [38, 39] (all open source) or Teradata [47] (closed source). This approach usually follows the Apache medallion architecture to produce domain-specific SQL-queriable table output sets. The current trend is that database query engines and distributed computing engines are increasingly becoming the interfaces for large-scale object storage, i.e the classical monolithic database is being unbundled [48, 49].

The data lakehouse storage tier often relies on a write-ahead transaction log-based architecture that operates as columnar data storage and supports ACID transactions. Technically, it is realised as a write-ahead log that tracks all atomic data changes, usually stored in parquet or JSON format. Each transaction log set for a particular table is accompanied by an expanding metadata set that stores schema, partitioning, and physical log data location information. Popular technology options for implementing this architecture include Apache Iceberg (initially developed by Netflix) [50], Apache Hudi (initially developed by Uber) [51], and the widely used Databricks-associated open-source Delta lake [19, 52]. Being based on transactional log data has the inherent benefit of providing out-of-the-box versioning and time-travel functionality.

For Data Lakehouse (or Delta Lakehouse if the storage tier is realised based on Delta table technology), Databricks [8] (deployable on all big cloud platforms) and recent entrant Microsoft Fabric [6] (deployed on Azure) are the dominant SaaS offerings that combine Apache Spark-based compute and Delta tables-based storage tier.[14] The most common minimal open source on-prem deployment pattern to create a delta lakehouse is to use dockerised Hive metastore instance [53] with MySQL (MariaDB) or Postgres-based Hive metastore storage, Minio S3 [54] object storage (for parquet files) and Spark [22] or Trino [38, 39] compute cluster. For a complete toy-example of creating a local delta lakehouse based on open-source technologies see also [55].

Decision dimension 3: *Selecting a deployment model, specifically between on-prem/IaaS and PaaS/SaaS cloud options*. It is important to note that these options are not mutually exclusive. On one end, many open-source or freemium data engineering tools allow free self-hosting with the option to upgrade to a paid managed deployment. Some examples of these relevant technologies that are also used in this thesis include Minio S3 [54] and Dagster [56]. Conversely, some of the most popular SaaS platforms are built on open-source technologies. For instance, Microsoft's Fabric data lakehouse platform utilises Apache Spark and Delta table open-source technologies [57].

One of the key advantages of PaaS/SaaS solutions, like complete data lakehouse offerings, such as Databricks and Microsoft's Fabric or orchestration and data movement-focused tools, such as Informatica [58] and Matillion [59], is the ease and security of getting the platform up and running. With built-in monitoring and no maintenance burden on the client company's DevOps team, it is a low-barrier way to introduce the technology into the company's tech stack. Additionally, these standardised platforms, supported by major tech companies, provide access to corporate training and upskilling initiatives. This results in a larger pool of potential external development partners and a more straightforward path to building in-house competencies.

The main drawbacks of the PaaS/SaaS data lakehouse platforms are twofold. They relate to high costs (see Appendix IV for short analysis of MS Fabric’s costs) and complexity in handling custom codebases and enforcing good programming practices.

To achieve the reliability and scalability comparable to cloud SaaS solutions with IaaS or on-prem deployment, companies must possess strong in-house DevOps skills and resources for. However, if a company already follows the infrastructure as a code practice and can deploy containerized software on a Kubernetes cluster, supporting modern DataOps stack should be feasible. Most open-source data engineering tools can run on IaaS or on-prem

[14] The MS Fabric suite also boasts a data warehouse storage module, which utilises the Synapse Data Warehouse SQL engine to facilitate the querying and transforming of data within Delta Lake (OneLake). This module offers full transactional capabilities and support for T-SQL flavour data definition language (DDL) and data manipulation language (DML) queries. However, it is important to note that only T-SQL can insert and update data within the data warehouse storage layer, while Spark and T-SQL can be used for querying. Additionally, it is worth mentioning that the underlying storage for this type of data warehouse is not an MS SQL server instance but rather still a set of delta tables. As a result, while it's technically classified as a data warehouse, it still serves as a partial data lakehouse implementation for teams with a skillset focused on traditional data warehouse platforms based on MS SQL (T-SQL dialect). The most significant functional difference between Fabric's data warehouse and data lakehouse is the former's support for multi-table transactions via the Synapse Data Warehouse SQL engine and the latter's support for non-relational data. Specifically, the architecture of the Fabric data warehouse is akin to the hypothetical situation of a Trino compute cluster running against Delta Lake, utilizing Hive metastore and S3 object storage. Further, as a storage tier, Microsoft Fabric also supports proprietary event- and streaming-focused databases called Eventhouse, which is queriable through the KQL dialect. The current thesis omits this novel technology branch [6].

Kubernetes clusters out of the box (e.g. [60]). Hybrid models are commonly used in large corporate settings, where SaaS solutions are implemented downstream of existing DW or data lake instances to enhance pre-reporting BI capabilities. In large corporate settings, the bulk of the custom codebase is likely deployed on-prem or on top of IaaS as pre-existing components, i.e. upstream from the SaaS platform. This approach lessens the main drawbacks of using a pure-play SaaS solution: costs and handling codebase complexity.

Decision dimension 4: *Choosing between the traditional coding paradigm and the no-code/low-code approach*. Supporting the leaning towards the latter is a fact that a significant portion of real-world tasks related to the extraction and movement of raw data are highly standardised. For instance, setting up a change data capture (CDC) stream from a relational database that saves atomic data changes to object storage; retrieving analytical business domain object data from popular services through API endpoints such as Salesforce (e.g. list of active campaigns), Jira (e.g. open projects and their progress), or Pipedrive (sales activity per client); or one-on-one copying relational tables from the live system to the staging area of the data warehouse. Writing and maintaining a custom codebase to accomplish previously described one-step data extraction and movement tasks would waste resources as other developers have already solved those problems in a reusable manner. All the previous tasks are accomplishable via a simple two-step GUI form-based workflow using an open-source Airbyte [61] or commercial Matillion [59] solution.

Deciding between low-code/no-code and traditional coding paradigms becomes more complex if the task assumes multistep and extensive custom data extraction or transformation operations. On the low-code/no-code side of things, not needing a programming background lessens the barrier of entry for the broader audience of the company's workforce. On the other hand, using GUI-based no-code/low-code tools might lead to problems with the long-term maintainability and refactorability of the data pipelines; as an example, see the discussion about the merits and drawbacks of heavily relying on PowerQuery for data transformations in chapter three.

Furthermore, on the higher level of problem complexity, even traditional procedural coding that is carried in a single or chained Jupyter notebooks might not suffice. Therefore, enforcing good programming practices is especially important if the analytics data management platform should, even partially, run the operations needed to extract analytic domain objects from the source system's transactional tables. As the example from VJS's (see chapter three of the thesis) showed, such code blocks can become quite long, i.e. thousands of lines, and the long-term maintainability of such codebase can become complex. Thus, adopting a standard project structure that will help decompose concerns into smaller testable code chunks is essential. Notebook and individual Python script-based data pipelines, which are the primary way to realise pipeline as a code on SaaS platforms, carry the risk of becoming another monolith. In the notebook context, implementing the proper coding etiquette takes extra effort. Empirically, the inclination of notebook medium to lead to a hard-to-maintain codebase is corroborated by the finding of Pimentel and colleagues [62] by which only 25% of Jupyter notebooks accessible in GitHub are reproducible. For possible remedies for this problem, see the subsequent article from the same group [63].

Decision dimension 5: When considering EVR's future analytic data architecture, it is essential to consider the *IT and business cooperation model in analytic data management*. This decision includes deciding between a centralised or data mesh [26] management model, which will define roles and responsibilities. While technical architecture is crucial, choosing the suitable data model (i.e. Data Vault 2.0, Inmon or Kimball) for storing analytic data depends mainly on domain specificity and the intended audience. Similarly, organising master and metadata management and assigning data owner roles depends on the level of expected involvement from the business side. Ultimately, the tools chosen for these tasks must reflect the desired (or practically achievable) level of collaboration between IT and business teams.

When managing analytic data in a centralised model, commonly used in traditional data warehousing, IT takes on the responsibility of providing comprehensive end-to-end analytic data provisioning service. Specifically, IT teams handle everything from obtaining business domain objects from transactional data in production systems to implementing data pipelines, managing meta and master data, and ultimately creating reports and organising data output through data marts. However, Dehgani [26] has outlined several challenges associated with this (corporate) approach:

1) When an external party, like the data warehouse development team, is responsible for building ETL pipelines on top of a live system's transactional tables after the business IT systems are already engineered and developed; the development process can become slow and resource-intensive. The situation arises because the data warehouse dev team is directly dependent on the ever-changing source data structure, which means that debugging and accommodating upstream changes can take up a significant amount of their resources. Essentially, the data warehouse acts as a partial anticorruption [24, 28] layer between the constantly changing IT systems and the stable warehouse data tier, similar to the role of the VJS's STATISTIKA_AGENT schema discussed earlier.
2) Deriving analytic data from operational data and the trade-offs and assumptions in obtaining the final ordered results can be complex and challenging for those outside the data warehouse development team. The problem is further compounded by the technical complexity of a monolithic data warehouse, which can obscure the data extraction and transformation logic. As a result, the data warehouse team may become the de facto owner and expert of the analytic data. Furthermore, the development process may experience a slow-down due to the need for the dev team to reverse-engineer all the specifics of domain logic before introducing new areas and functionalities to the central provisioning analytic data.
3) The end user, who exhibits the highest domain knowledge of the analytic data, is relegated to the passive consumer of the cleaned data and reports.

While the centralised approach may have drawbacks, it still has practical applications, especially for large corporate settings. It is essential to consider the resources required to implement the full-service model carefully. Additionally, many organisations create new roles, like data stewards, to facilitate understanding between the data warehouse team and business-side data consumers. Furthermore, anticipating the data warehouse's expected input during the requirements engineering phase, documenting it, and maintaining its stability can mitigate the risks associated with the centralised approach.

The innovative data mesh approach [26] differs from the traditional centralised method of analysing datasets in a linear fashion. Instead, the data mesh model proposes that large and intricate datasets should be divided into decentralised data domains and managed by self-governing cross-functional teams. These teams consist of individuals with domain expertise and essential IT skills who should be aligned with specific business or functional areas to develop more in-depth expertise and ownership over their data. The fundamental concept of data mesh is to treat data as a product. Product-minded thinking means that domain-specific datasets should be created in a way that seeks to provide value for other domains, ensuring that published data is independently discoverable, usable, and combinable by other teams within their domain with minimal effort. Additionally, domain-specific datasets should have explicitly stated quality standards and service-level agreements (SLAs), typically formulated as data contracts (for a practical template of a data contract, see [64]).

To illustrate the technological shift brought about by the data mesh approach, one could draw a comparison to the domain-driven microservices architecture design that revolutionised complex monolithic software [24, 26]. This analogy has multiple compelling connotations. Firstly, data teams should learn from the microservices world how to manage the data schema drift, i.e. by adopting the publishing model of analytic data with the possibility of accommodating data structure change by creating an alternate version of the published datasets. Secondly, like microservices architecture, data mesh can only succeed if the autonomous teams implementing and servicing their domain-bounded data publishing microservices are technologically and architecturally empowered to work independently without being strongly coupled to upstream dependencies.

Microsoft's latest SaaS data engineering and reporting solution, Fabric, is an excellent example of this. By promising to seamlessly integrate the Delta Lake platform, similar to Databricks, with pre-existing low or no-code data extraction and transformation capabilities (now merged under DataFlow Gen2) from Power BI and Data Factory, as well as reporting functionalities from Power BI, Fabric has emerged as a first big platform that tries technologically operationalize the complete data mesh concept. Of note are the direct references to the data mesh concept in Fabric's documentation [65, 66], which again define Fabric's role as an enabler of company-wide data culture by making data mesh a practical possibility.

All in all, Fabric's most advertised standout feature is not a specific new narrow technology or capability but rather the amalgamation of various technologies onto one platform that should allow domain-oriented teams to create and utilise analytic data as a product. The new platform has significantly invested in low/no code tools and a shared visual interface to foster a collaborative development environment across users with differing technical competencies. In detail, the new platform tries to cater to professional report developers using Power BI, data engineers utilising the Spark notebook interface and Spark job definitions, and self-serve users, also known as citizen developers by Microsoft, who rely on Power BI reporting and PowerQuery-based data transformations.

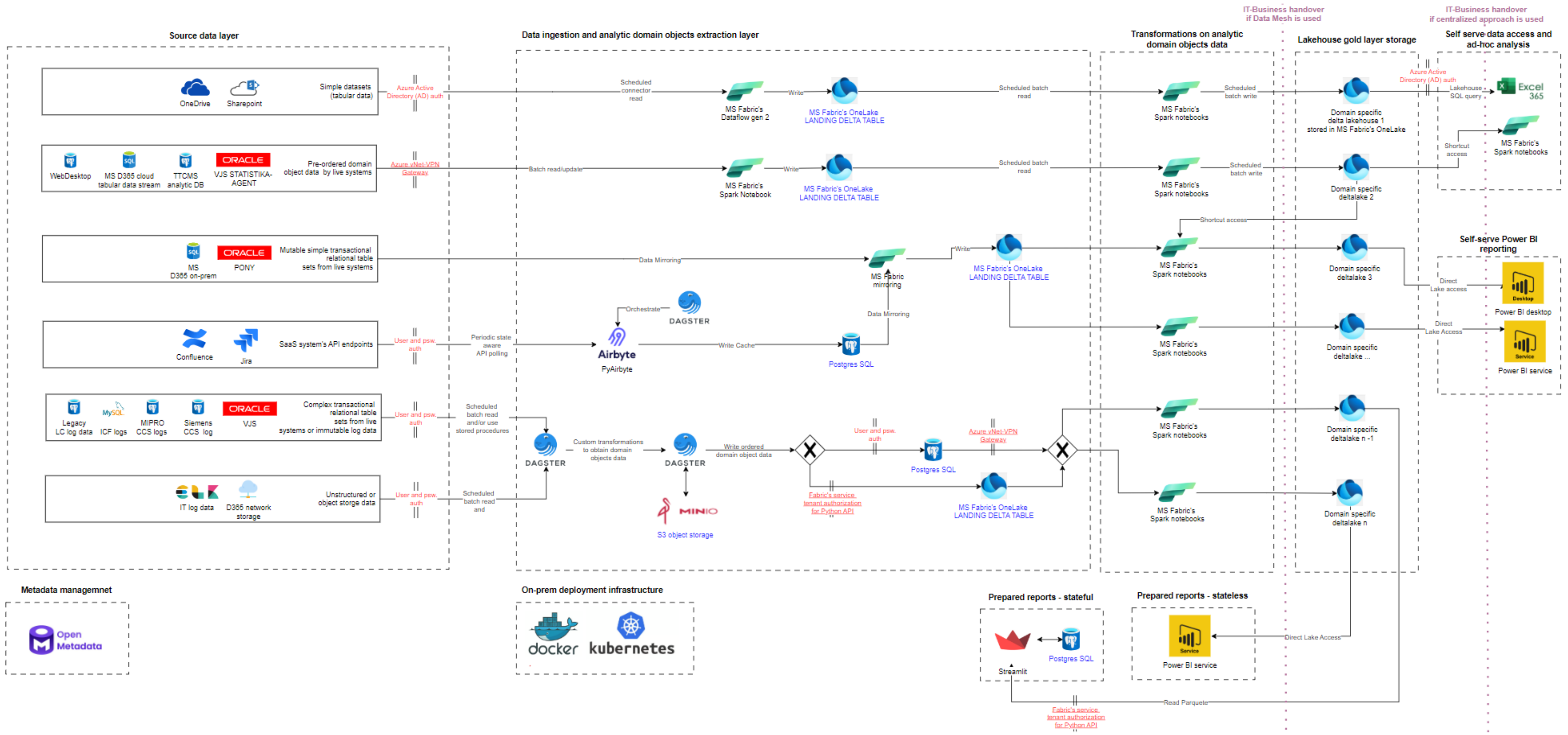
4.2 The Proposed Technical Architecture for EVR's TO-BE Analytic Data Management System

This thesis section outlines the proposed TO-BE state for analytic data management in EVR. First, a visual representation of the general architecture (see Figure 7) and a brief overview of its key components is offered. Additionally, the proposed architecture is exemplified through potential implementation use cases, each with varying levels of complexity, all of which are based on the TO-BE architecture.

The fundamental tenet of the proposed architecture is the pivotal role assigned to Microsoft's new SaaS Delta Lakehouse Fabric. At the time of writing this thesis, EVR is preparing a proof-of-concept study in collaboration with Microsoft's cloud services partner company to evaluate the compatibility of Microsoft Fabric with EVR's business needs. Given that Power BI is tightly integrated with Microsoft Fabric, it is the natural first choice for a technology trial, as the current EVR's analytic data management and reporting system has been developed around this tool (refer to chapter three for further information). Furthermore, the practical use cases of the thesis will be used as the basis for the proof-of-concept study.

Therefore, TO-BE architecture delves into integrating Fabric with existing on-premises data sources and solutions and the optimal balance between Fabric and other data engineering-focused on-premises or cloud technologies. It aims to provide a structured analysis of these open questions to aid in selecting the most suitable capabilities from Fabric's total offering. Additionally, the TO-BE architecture advanced by the thesis explores ways to enhance Microsoft's SaaS offering with open-source technologies that serve as the integration middle layer between Fabric and on-prem business IT systems. It is worth noting that since the initial public preview of Fabric in the summer of 2023, Microsoft has revamped and changed its offering quite a bit, with features being added and removed. Therefore, it is essential to remember that the current way of incorporating Fabric and its capabilities in the TO-BE architecture is based on the form of the Fabric platform at the time of writing the thesis.

In the larger context, the introduction of Fabric will probably result in a gradual phase-out of Microsoft's previous “all-in-one” data integration solution, Azure's Synapse Analytics, which may come as an unwelcome surprise for companies who have invested in building their analytic data management system using this platform. Thus, incorporating additional technologies and utilising a hybrid deployment model alongside Microsoft's solution provides the welcomed benefit of reducing the risk of vendor lock-in by making the entire architecture more flexible for potential extension with competing alternatives. One example is the TO-BE architecture's ability to integrate Amazon EMR's Spark compute cluster (via Dagster) as an option with a more affordable and adaptable per-time-used pricing structure, as opposed to Fabric's costly and inflexible upfront pricing model. In other words, the hypothetical Amazon EMR's usage showcases the usefulness of preserving a certain level of on-premises or IaaS infrastructure in one's configuration, which retains the liberty to select tools from the deployment spectrum ranging from IaaS to SaaS.



IT-Business handover if Data Mesh is used
IT-Business handover if centralized approach is used
Source data layer
Data ingestion and analytic domain objects extraction layer
Transformations on analytic domain objects data
Lakehouse gold layer storage
Self serve data access and ad-hoc analysis
OneDrive
Sharepoint
Simple datasets (tabular data)
Azure Active Directory (AD) auth
Scheduled connector read
MS Fabric's Dataflow gen 2
Write
MS Fabric's OneLake LANDING DELTA TABLE
Scheduled batch read
MS Fabric's Spark notebooks
Scheduled batch write
Domain specific delta lakehouse 1 stored in MS Fabric's OneLake
Azure Active Directory (AD) auth
Lakehouse SQL query
Excel 365
WebDesktop
MS D365 cloud tabular data stream
TTCMS analytic DB
ORACLE
VJS STATISTIKA-AGENT
Pre-ordered domain object data by live systems
Azure vNet-VPN Gateway
Batch read/update
MS Fabric's Spark Notebook
Write
MS Fabric's OneLake LANDING DELTA TABLE
Scheduled batch read
MS Fabric's Spark notebooks
Scheduled batch write
Domain specific deltalake 2
Shortcut access
MS Fabric's Spark notebooks
Shortcut access
MS D365 on-prem
ORACLE
PONY
Mutable simple transactional relational table sets from live systems
Data Mirroring
MS Fabric mirroring
Write
MS Fabric's OneLake LANDING DELTA TABLE
MS Fabric's Spark notebooks
Domain specific deltalake 3
Self-serve Power BI reporting
Direct Lake access
Desktop
Power BI desktop
Orchestrate
DAGSTER
Data Mirroring
Confluence
Jira
SaaS system's API endpoints
User and psw. auth
Periodic state aware API polling
Airbyte
PyAirbyte
Write Cache
Postgres SQL
MS Fabric's Spark notebooks
Domain specific deltalake ...
Direct Lake Access
Service
Power BI service
Legacy LC log data
MySQL
ICF logs
MIPRO CCS logs
Siemens CCS log
ORACLE
VJS
Complex transactional relational table sets from live systems or immutable log data
User and psw. auth
Scheduled batch read and/or use stored procedures
DAGSTER
Custom transformations to obtain domain objects data
DAGSTER
Write ordered domain object data
User and psw. auth
Postgres SQL
Azure vNet-VPN Gateway
MS Fabric's Spark notebooks
Domain specific deltalake n -1
IT log data
D365 network storage
Unstructured or object storge data
User and psw. auth
Scheduled batch read and
MINIO
S3 object storage
Fabric's service tenant authorization for Python API
MS Fabric's OneLake LANDING DELTA TABLE
MS Fabric's Spark notebooks
Domain specific deltalake n
Metadata managemnet
Open Metadata
On-prem deployment infrastructure
docker
kubernetes
Prepared reports - stateful
Streamlit
Postgres SQL
Prepared reports - stateless
Service
Power BI service
Direct Lake Access
Read Parquete
Fabric's service tenant authorization for Python API

Figure 7. Outline of the TO-BE analytic data management architecture.
Notes: Vertical purple lines denote the work handover points depending on the implemented IT-business cooperation model.

The proposed architecture is based on the premise that source data of varying complexity, such as raw transactional data versus pre-cleaned and aggregated data, necessitates distinct data pipelines that may also vary in terms of the technologies employed. Figure 7 categorises EVR's data sources across six groups:

1) The first group includes simple tabular datasets with a limited number of tables and no complex, multilevel depth relationship between them. These datasets can be stored as Excel or CSV files on Microsoft's OneDrive or sourced from SharePoint site's embedded tables.
2) The second group includes relational analytic data, which already contains analytic domain objects that have been fetched from live transactional tables by the source system. Further, as exemplified by the VJS's schema STATISTIKA_AGENT, the domain objects can be pre-aggregated, e.g. by period and client. In the near future, EVR's new traffic management system TTCMS, refactoring of infrastructure cost calculations system (extension to VJS's STATISTIKA_AGENT) and the cloud version of Dynamics 365 through Dataverse's tabular data stream will all produce similar pre-ordered analytic domain objects data.
3) The third category comprises relational source data table sets that can be leveraged to acquire analytic domain object data through straightforward joins and the SQL query engine's built-in grouping and aggregation functions, which obviates the necessity of creating or using custom SQL procedures/functions from the source database. Additionally, the table sets required to derive an analytic domain object with its attributes are easily manageable, with a maximum size of 10-20 tables, mainly limited to 3-7 tables.
4) The fourth category comprises data from API endpoints of cloud-based SaaS solutions. In the case of EVR, Jira and Confluence are the most critical SaaS solutions, which should provide operative analytic data output to EVR's delta lakehouse (Fabric) and reporting (Power BI) layers.
5) Within the fifth category, the relational source data is comprised of transactional tables that cannot be readily utilised to derive analytic domain objects via the SQL query engine's native grouping and aggregation functions. Custom procedures within the source system database must be employed to obtain the full analytical domain objects, or a significant amount of custom programmatic logic must be developed. A prime example of the latter scenario is log data that necessitates algorithmic interpretation, such as aggregation and pattern matching, to acquire higher-order log event cascades that carry business significance. As an illustrative instance, a train passing through a level crossing involves a set of atomic events from various underlying technical systems, which must be algorithmically grouped together to obtain the analytically meaningful train passthrough event.
6) The sixth category of source data encompasses unstructured or non-relational object storage data. Similar to the last category, a considerable amount of custom program logic, including parsing and fetching, is necessary to extract meaningful analytic domain objects from this data.

Based on the input data, the architecture developed by the thesis proposes various intermediate processing layers between the source systems and the Fabric cloud platform. However, the final Delta Lake (gold) storage and reporting layer is consistent across all six data sources and is expected to be fully implemented on Fabric SaaS service. To elaborate, for the final layer, which is identical for all six data sources, data is first imported from staging/landing tables. Then, on the Fabric platform, Spark Notebooks are used to transform the data into cleaned and dimensionalised datasets, which are stored in domain-specific (e.g.

finance, infrastructure) gold layer delta lakes (i.e. collections of linked domain-specific delta tables). These gold-layer Delta Lakes serve as the foundation for self-service data access, analysis, and reporting and for centrally developed stateless and stateful reports. As a reminder, statefulness in the context of reports implies that reports must support basic CRUD operations for a set of analysis parameters.

It is worth noting that the level of complexity in the final Fabric-based layer of transformations is variable based on the source data. Further, this deliberate variability in implementing the middle layer between the source data system and the Fabric platform allows flexibility in adjusting the costs of the overall architecture. For example, tasks that require more data and computational power to extract analytic domain objects from transactional data can be accomplished on less expensive on-prem/IaaS or PaaS infrastructure. Meanwhile, computationally cheaper transformations on analytic data can stay reserved for the pricier SaaS-level solution. Overall, the architecture developed by the thesis allows attenuating the maximum complexity before the costly reporting gold layer storage layer.

The first data category (refer to nomenclature in the previous page) requires no preliminary on-prem/IaaS preparatory layer, which involves straightforward tabular data stored on Microsoft's cloud platforms (SharePoint or Office 365 OneDrive). Technically, Fabric can pull this data source type directly into staging delta tables via Dataflow Gen 2 in-built connectors, followed by the shared Fabric-based gold storage and reporting layers described earlier for all source datatypes. Semantically, most simple tabular datasets stored in SharePoint or OneDrive already describe analytic domain objects. Therefore, there is no need to implement steps for extracting analytic domain data objects. An example of this pattern in action would be KPI indicators stored in embedded tables on a SharePoint site.

When the source system provides clearly defined analytic domain objects, as is the case for the second category, a preparatory layer is unnecessary, just like in the first category. Batch read jobs (using the pull architectural pattern) read data into staging Delta Tables for ingestion into the Fabric platform using Spark Notebooks. The final stages of the data pipeline follow the same shared pattern as before. For example, pre-aggregated data tables from VJS's STATISTIKA_AGENT schema needed to calculate infrastructure usage fees only require a minimal transformation (if any) to perform as a suitable data source for the reporting layer. Therefore, the most optimal solution is sourcing the data with the same frequency as the source system produces it (once per day) by initiating pull queries from Fabric.

To enable the analysis of domain objects and their relevant characteristics, the third and fourth categories of data sources rely on a select number of transactional tables from rapidly evolving source systems. In order to achieve a near-real-time reflection of the source system's state within Fabric's staging tables, the proposed architecture utilises a change data capture (CDC) or mirroring technology pattern. Fabric's data mirroring functionality is used for relational data sources. For the fifth data category (SaaS System's API endpoints), data mirroring is carried out by the no-code open-source tool Airbyte's minimised Python library PyAribyte that is orchestrated by Dagster [67].

The previously described data extraction patterns constitute a push architectural pattern, as changes in the source system are automatically propagated to the target (Fabric staging Delta Tables) through the streaming link. For this middle layer configuration, Fabric's Spark notebooks extract the analytic domain objects from the source system's tables and, per the standardised following, store them in the gold layer Delta Tables. Of note, as outlined above in the description of data sources, the delegation of extracting analytic domain objects from transactional tables to the Fabric should be only under the constraint of analytic domain objects being extractable through simple(r) SQL joins and ISO SQL-standard group- and aggregate functions, from not overly excessive table set. The illustrative use case would be reporting on work permit issuance and statuses based on the data stored in the PONY system (small module in PONY). As PONY's data model is near Inmon and most of the data-logical operations performed consist of simple CRUD (create, read, update, delete) transactions, live-mirroring those tables to the Fabric via structured streaming link is reasonable. Obtaining reporting source data from those mirrored tables would be a straightforward task.

For the last two categories of data (unstructured or object data and complex relational transactional data or immutable atomic log data) that demand intricate and resource-intensive multistep logic to extract analytic domain objects, the TO-BE architecture recommends constructing pipelines with the assistance of Dagster. Dagster is a data orchestration tool that follows a development and project structure akin to conventional software development practices, as opposed to the definition of data pipelines using notebooks. Therefore, Dagster plays a unique and non-redundant role in TO-BE architecture compared to Fabric's Spark notebook-based development interface. An excellent use case for the domain object extraction layer described earlier would be generating reports based on the Soviet statistical system. To provide some context, EVR must operate two separate statistical reporting systems. The first is the so-called Russian system implemented under the Council for Railway Transport of the Commonwealth Member States (CSZT), which covers Soviet 1520mm track gauge railways from the Baltic States, Russia, Poland, and other countries. The second is an internal system similar to the European reporting system. These two systems are largely incompatible; even the main domain events (such as train entity reckoning) have differing implementations. The Soviet or CSZT reporting system relies on VJS's UI views, which are supported by procedures and queries executed directly, i.e. without STATISTIKA_AGENT-like standardising middle layer, against VJS's internal transactional tables and procedures. Developing a stateful domain object middle layer similar to STATISTIKA_AGENT for select Soviet system reporting views would be a beneficial application to implement through Dagster.

The middle layer of the TO-BE architecture, consisting of Minio S3, Dagster, Airbyte (either as Dagster orchestrated Python package or standalone installation) and Postgres, and basic CRUD reporting capabilities using Streamlit, can be easily deployed on-prem or through IaaS with Docker. This option allows deployment on either a single node or a Kubernetes cluster.

Additionally, effective metadata management is a crucial component of the TO-BE architecture that can help bridge gaps in capability, such as improving documentation and data discoverability, as outlined in Table 1.0. For the cloud, i.e. Fabric, side of the architecture, data lineage and metadata management is already built into the SaaS offering. For comprehensive data lineage management across the source data layer and middle layers, which encompass ingestion and analytic domain object extraction, TO-BE architecture recommends utilising the open-source tool OpenMetadata, which is a user-friendly, centralised repository for managing metadata [68]. It provides a UI-based environment to automatically derive, store and update information about data assets, including their lineage and attributes. Importantly, OpenMetadata enables management and cataloguing of relational data assets at the table, column, and stored procedures level. It also allows users to assign data owner roles and create personalised data categorization descriptive metadata (data dictionary).

The implementation of OpenMetadata has the potential to solve multiple challenges. Firstly, it enables the documentation of dependencies between transactional data in the source system and the analytic ingestion level, simplifying identifying and resolving issues related to upstream dependencies. Additionally, an OpenMetadata-like environment can facilitate the development and maintenance of source systems to improve documentation coverage for data assets, resulting in greater data discoverability and more swift extraction of analytic data from transactional sources. Appendix II of the thesis presents an initial TO-BE blue-print for metadata management and data product-based IT-business cooperation pattern.

5 Practical Use Cases Based on Selected Components of TO-BE Architecture

The chapter presents a detailed walkthrough of implemented two end-to-end use cases based on TO-BE architecture: the first case involves the comprehensive analysis of row-level data from purchase invoice XML files. This analysis includes data fetching, cleaning, dimensional normalisation to a star schema, data enrichment, and reporting from a Postgres-based data warehouse. The second use case revolves around the end-to-end analysis of railway level crossing log data, which involves handling unordered source data through algorithmic data mining and reporting. Both use cases are implemented following a dockerised and version-controlled deployment model.

The reason behind choosing those two particular use cases is that the current EVR's analytic solutions have the least amount of previous experience in dealing with unstructured data sources and the analytic data ingestion and domain object extraction layer is technically the most complex (refer to Figure 7) for those two source data types. Thus, the decision was made to trial out the new architecture on those two analytic areas. Further, automatic analysis of log data and its integration into business processes is still a relatively nascent development for the company. Its business potential is emphasised, particularly in improving maintenance schedules and overall railway safety. While the current Power BI-based analytics solution has successfully addressed reporting challenges based on pre-aggregated or narrower datasets with low data volumes and liberal update schedules (predominantly once a day), the decision to trial the proof of concept on newer and less familiar datasets is a riskier proposition. Nonetheless, the potential for in-house technological competency building justifies this approach.

The code developed by the author is available at [69].

5.1 Deployment Context of Implemented Practical Use Cases

Considering that the data pipelines operate on live data within the company's access network, thinking through deployment networking, security, and authentication was paramount. The VM running the staging server is deployed behind the firewall (i.e., no ports open to the outside world), has a static IP address, and internally mapped DNS record. Further, the deployment context was sandboxed through custom-rules-based VLAN, limiting its access only to needed source IPs and external software repositories. Additional networking rules, e.g., the decision to which Docker container ports to expose outside of the host machine and which ports to keep exposed only at the host machine's host level, were made at the level of Docker-compose configuration parameters.

Regarding the software, the staging server runs Oracle (CentoOS) Linux [70] with *yum* package manager. Deployment was carried out in a single-node context by setting up the Docker engine [71] with the Docker-compose plugin [72]. Figure 8 describes the deployment context in which both POC use cases were realised. Firstly, of importance, in terms of the last figure, is the general network architecture of the setup, which also includes signalling patterns between Dragster containers and source data resources. Secondly, Figure 8 illustrates the role of four Docker containers that both of the use-case depend on:

1) *Dagster daemon* container - plays a central role in a Dagster deployment, especially in container-based orchestration systems like Docker or Kubernetes. This container is responsible for executing sensors and schedules, managing run queues and monitoring run execution. Of note is the possibility of running Dagster daemon, Dagster UI, and working code in the same container by creating a Python virtual environment (*.venv*) runtime for Dagster. This pattern is often used for local development, which shows Dagster's flexibility of not being rigidly dependent on deployment infrastructure. However, for more complex use cases, such as realised by the thesis, it is recommended to use a multi-container setup (*Dagster daemon*, *Dagster UI*, storage and one or more *worker deployment* containers), which helps in discernment of concerns, code maintainability and future scalability.
2) *Dagster dagit* (UI container) - this container runs Dagster webserver, which provides basic UI for the containerized deployment setup. Working code execution context (worker or deployment containers) are triggered via gRPC-based remote process calls. Worker container statuses are fed back to Dagit via API endpoint (e.g. deployment state or pipeline execution finished) or via queries to Dagster logs and metadata containing database container.
3) *Dagster logs and execution metadata storage container* in the form of Postgres 14-alpine Docker container - this container stores event logs, schedules and run storage (i.e. metadata describing pipeline runs execution context). As most of the Dasgter documentation and most often referenced example repos used Postgres 14 as an Dasgter log storage layer, similar decisions was taken by the thesis.
4) *Temporary data warehouse* in the form of Postgres 16-alpine Docker container. This container stores the data that is used as input to the realised reporting, Power BI Desktop report for purchase e-invoices XMLs and Streamlit dashboard for level crossing log data.
5) Minio S3 container – for object storage. This container is used to enhance the statefulness of data pipelines (see section 5.2 for details).

All the other elements (sources, outputs and other Docker containers) from the Figure 8 are use-case specific and are thus described in the respective sections.

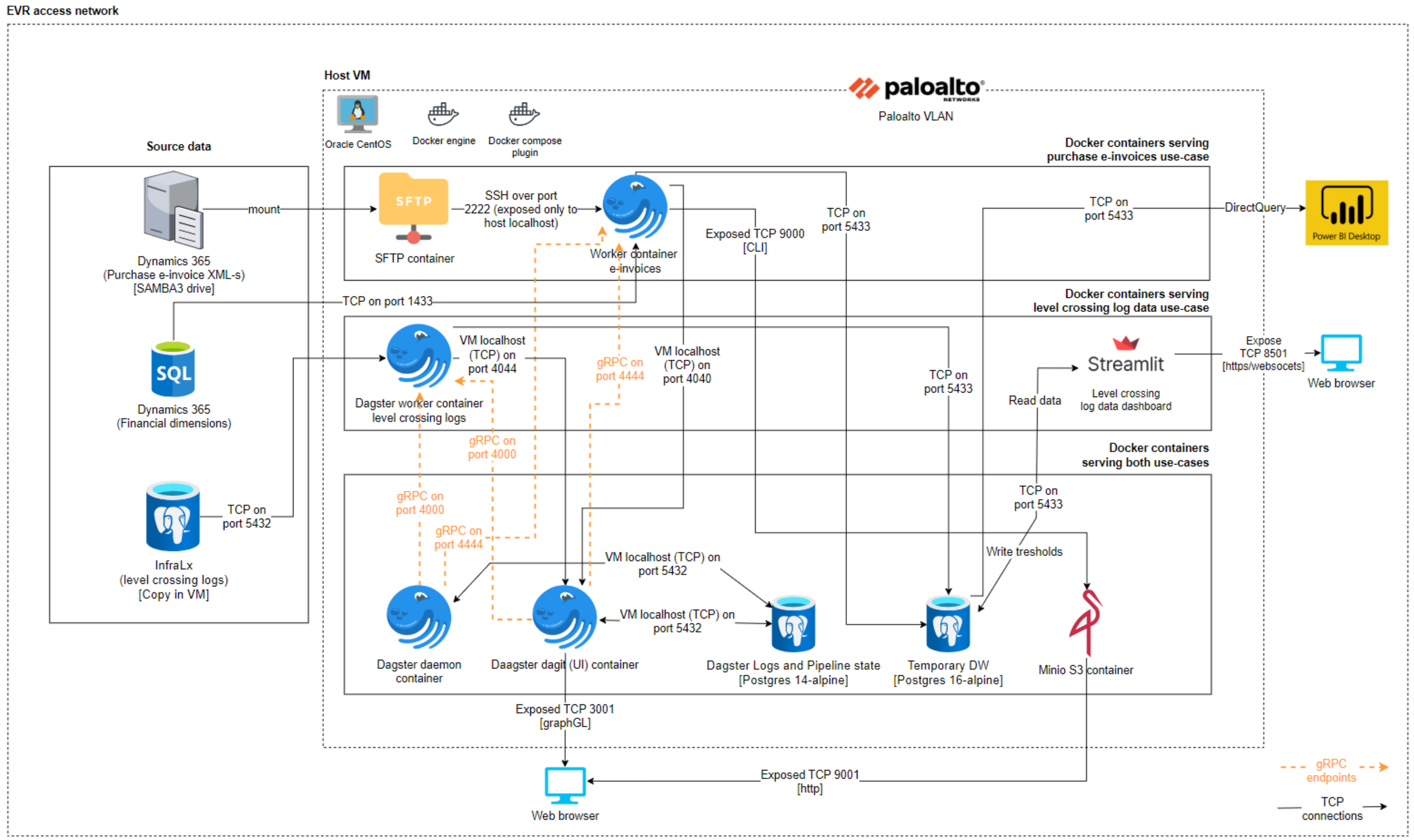

EVR access network
Host VM
paloalto NETWORKS
Paloalto VLAN
Oracle CentOS
Docker engine
Docker compose plugin
Source data
Dynamics 365 (Purchase e-invoice XML-s) [SAMBA3 drive]
mount
SFTP
SFTP container
SSH over port 2222 (exposed only to host localhost)
Worker container e-invoices
Exposed TCP 9000 [CLI]
TCP on port 5433
Docker containers serving purchase e-invoices use-case
TCP on port 5433
DirectQuery
Power BI Desktop
TCP on port 1433
Dynamics 365 (Financial dimensions)
Docker containers serving level crossing log data use-case
VM localhost (TCP) on port 4044
Dagster worker container level crossing logs
gRPC on port 4444
VM localhost (TCP) on port 4040
TCP on port 5433
Read data
Streamlit
Level crossing log data dashboard
Expose TCP 8501 [https/websocets]
Web browser
gRPC on port 4000
Docker containers serving both use-cases
InfraLx (level crossing logs) [Copy in VM]
TCP on port 5432
gRPC on port 4000
gRPC on port 4444
VM localhost (TCP) on port 5432
TCP on port 5433
Write tresholds
VM localhost (TCP) on port 5432
Dagster daemon container
Daagster dagit (UI) container
Dagster Logs and Pipeline state [Postgres 14-alpine]
Temporary DW [Postgres 16-alpine]
Minio S3 container
Exposed TCP 3001 [graphGL]
Exposed TCP 9001 [http]
Web browser
gRPC endpoints
TCP connections

Figure 8. Deployment context.

5.2 End-to-end Row-level Analysis of Purchase Invoices from Raw XML-s

The initial use case pertains to fetch and enrich row-level data from purchase e-invoices XML-s. In Estonia, suppliers are mandated to provision e-invoices [73], and the e-invoice format has been standardised [74, 75]. Ordinarily, the e-invoices are transmitted from suppliers to purchasers through an invoicing service provider (such as Finbite/Omniva) intermediary, which also conducts the journalization of the e-invoices against the journal account schema of the purchasing entity, utilising a blend of methods, both manual and automated.

EVR's e-invoices are received and consolidated onto an on-premises network drive that is also responsible for managing additional attachments essential to the day-to-day operation of EVR's Dynamics 365 enterprise resource planning system. These attachments may include photographs, XML files (such as messages from invoicing intermediaries detailing the general account journaling of raw vendor-produced e-invoices), Word and Excel documents, and other related materials. Notably, all these documents are stored on the network drive as serialised file blobs, bearing hash-based filenames and devoid of file extensions.

The initial technical obstacle is retrieving the e-invoice XMLs securely. Additionally, given that the network drive has accumulated almost five years' worth of attachment history, the retrieval process must be carried out without overburdening the VM-based server responsible for serving the on-prem network drive, which also serves the live system. This technical challenge was addressed by implementing the following steps:

1) New Microsoft's ActiveDirectory's read-only user was created with access rights to the network drive housing the D365's file attachments.
2) The SFTP Docker container was set up based on the *atmoz/sftp* [76] image, and the D365's network drive was mounted to the SFTP container as an external volume.
3) The SFTP container was port-mapped to be exposed only to the address space of Docker's host machine VM IP address to improve security.
4) Next, Dragster worker containers and Minio S3 object storage containers were set up. In the Dagster worker container, two resources[15] were generated by extending the Dagster's base I/O class [78]. Firstly, *pysftp* [79] package-based I/O resource to SFTP Docker container and secondly, *minio* python package-based [80] I/O resource to Minio S3 object storage.

[15] In Dagster, a resource is a configurable object that represents an external asset or service, such as a database connection or a cloud storage client. Resources provide a way to encapsulate setup and teardown logic for these external assets, making it easier to manage their lifecycle within your data pipelines. Thus, the asset allows for abstracting away the details of connecting to and using these external services making the pipelines more modular and reusable, as the same resource can be used across multiple pipelines or pipeline components without needing to duplicate the connection logic. [77]

5) The first Dagster's asset[16] *get_files_to_parse* was generated based on the previously created resources. This asset first lists (name metadata, not files themselves) all the files from the last three days (intra-asset configurable parameter) that are stored in the D365 on-prem network drive. Secondly, it accesses JSON logs that are generated during the pipeline activity to check if the serialised file blob is a valid e-invoice XML. The Dagster asset generates a Python list containing the set difference of available and already checked files. Importantly, to enhance the confidentiality of the data handling, only files smaller than four megabytes are handled (configurable intra-asset parameter), which reduces the load on the source data platform and partially mitigates accessing files that contain non-XML confidential information (e.g. sizable invoice related attachments).
6) Next, a Dagster data asset called *get_new_XMLs* is created to process a list of unchecked new XMLs. This asset takes as input a list of unchecked new XMLs generated by the *get_files_to_parse* asset. To ensure that the XML is valid, a custom script is used to parse the candidate serialised file blob as an XML. The script uses the Python library *lmlx* [81] to parse the files and checks if the XML file is an instance of an e-invoice through a simple *xpath* query. The files are accessed individually through an SFTP-accessing resource class method call, while the read-in file is stored in the Dagster worker node's memory. All the per-source file checks and valid e-invoice XML file hits are logged to Minio S3 logging buckets. This prevents the subsequent reruns of the data pipeline from carrying out redundant work, such as checking already checked files. The *get_files_to_parse* asset generates a return dictionary, with keys defined by the source file name and its values represented by *etree*'s UTF-8 encoded string dumps of the XML. Dumping parsed XML as a string in the output is required, as Dagster cannot auto-pickle l*mlx*'s *etree*-specific data class.

The following major technical challenge is extracting row-level data from valid purchase e-invoices XML-s and enriching the row-level data with relevant financial dimensions. This technical challenge was addressed by implementing the following steps:

1) A Dagster data asset *XML_to_results_df* is created. Based on the custom script, this asset tries to fetch the following values for each of the invoice lines from the valid purchase XML-s:
    a. Seller name.
    b. Seller registration number.
    c. Invoice number.
    d. Invoice date.
    e. Purchased item amount.
    f. Item unit.
    g. Description of the purchases item.
    h. Price reduction (*AddRate*) rate.
    i. Per item pierce (*ItemPrice*).
    j. Item total (usually item amount times per item price).

[16] In the context of Dagster, an asset represents a piece of data or a data-related computation. Assets in Dagster manage how data is produced, state-managed, and monitored. Each asset is defined by the computation that produces it, along with metadata that describes its dependencies on other assets, its output data type, and its lineage. Dagster allows assets to be materialised - that is, physically saved or updated in a database, local drive or data lake. Further, Dagster allows for automatic pickling of asset output (even if its base Python object like dictionary or list or Pandas' or Polars' dataframe). [82]

k. Item’s EAN (European Article Number) number.
l. Seller’s product ID.
m. XML filename.

The script to retrieve row-level data from valid purchase XMLs is based on a combination of *xpath* queries and Python dictionary data structure-based search. When a XML tag is missing and cannot be found by the fetching algorithm, the script returns a *Missing Tag* string value for the corresponding purchase e-invoice line. Similarly, suppose there are multiple matches with differing values for the same tag present in different XML hierarchy levels. In that case, the script returns *a Multiple Values* string value for the corresponding purchase e-invoice line.

If an error occurs during the fetching process, the exception is captured and the corresponding XML file is registered in the Minio S3 bucket named *xml-to-df-errors*. This follows an architectural pattern where problematic files are handled in a separate data pipeline from the regular (happy) path described herein. However, the pipeline for resolving errors has yet to be implemented.

The data pipeline under consideration includes various assets, and among them, the most troublesome one is the *XML_to_results_df* fetching step. This is because the Estonian e-invoice XML standard [75] is quite liberal, and creating a universally applicable algorithm within the confines of the current thesis has proven to be nearly impossible, given that vendors tend to disregard the standard to some extent. The most common errors include storing information that should be contained in an XML tag as an XML tag attribute or in an unrelated generic information content tag, using custom fields, and saving inappropriate information in XML tags (e.g., providing buyer contact information as item entry). To address these issues, vendor-specific fetching rules will need to be developed in the future.

The data *XML_to_results_df* asset returns a dictionary whose keys are defined by purchase invoice XML names and values by Pandas data frames containing the fetched rows of the corresponding purchase e-invoice.

2) The subsequent branching data asset *result_DFs_to_S3* saves the invoice rows data frames from the previous step to the Minio S3 bucket as JSON files (one JSON file per previously fetched XML). This data asset constitutes one potential handover point between the on-prem system and the MS Fabric cloud data lakehouse. Based on test deployment outside of EVR’s infrastructure, on the Hetzner cloud infrastructure, it was possible to stream an on-prem S3 bucket to the data lakehouse’s delta sink table via Spark structured streaming.
3) In another branch, data asset *set_up_dw_xml_landing* sets up a gold layer data landing schema, tables, procedures and triggers in previously referenced Postgres 16-alpine Docker container-based temporary DW. If required elements are already present in the DW, the step is skipped at the database level, i.e. the Postgres DW I/O Dagster resource calls a Postgres’ parameterized table, procedure and trigger creation SQL script templates, kind of a poor man’s *dbt* equivalent, that include either try-catch logic or queries against Postgres’ system tables.

If not already present, the data asset creates the following 13 tables based on the template of the star schema:

a. Fact table *fact_invoice_xml*

As the missing tags (not found from XML by fetching algorithm) and multiple matchings (the same tag is present in multiple XML hierarchy levels with differing values) are returned as string outputs, the decisions was taken, including the consideration of facilitating quick debugging, to keep most fact table fields as varchar type.

Dimension tables [17]:

b. *dim_accoun*t for account;
c. *dim_contract* for contract;
d. *dim_cost_center* for cost centre;
e. *dim_department* for department;
f. *dim_main_asset* for main asset;
g. *dim_project* for project dimension.

The six n-m logic tables between the fact table and dimension tables are essential, whose naming follows the convention of *fact_invoice_xml_x_[respective dimension table name]*. Those n-m tables exist because financial dimensions from MS SQL DB can be matched with specific XML only at the whole purchase e-invoice level. In other words, there is no direct way to match rows of purchase invoices with specific financial dimensions. There is a possibility for fuzzy matching (not implemented) based on bookkeeping entries and related sums, but it is computationally a complex problem which assumes almost perfect row-level data fetching from XML-s. Therefore, as it stands now, all the related financial dimensions for a given purchase e-invoice XML are related to each fetched row of the given purchase invoice.

Figure 9 depicts the E-R diagram of the output data of the whole data pipeline.

4) The data asset *load_fact_table_to_dw* upserts (insert if id is not present; update if id present) fact table entries from the data asset *XML_to_results_df* to Postgres temporary DW. The following concatenation pattern is used to generate the surrogate keys: *[invoice row order]_[invoice number]_[source XML filename]_[date]*.
5) The data asset *get_dims_to_load* uses *mssql_io_manager* resource to fetch financial dimensions that have been related to the given e-invoice in the Dynamics D365 ERP MS SQL database. Based on Microsoft's documentation [83] and initial consultations with EVR's internal product owner, it was possible to work out the associated financial dimensions by parameterizing SQL query for invoice number and seller registration code. The dimension fetching query encompassed the following tables and materialized views from the D365 ERP system:
   a. *DCEINVOICETABLE*.
   b. *GENERALJOURNALENTRY*.
   c. *GENERALJOURNALACCOUNTENTRY*.
   d. *DIMENSIONATTRIBUTEVALUECOMBINATION*.
   e. *ASSETTABLE*.
   f. *DIMATTRIBUTEOMCOSTCENTER*.
   g. *DimensionFinancialTag*.
   h. *EVRCONTRACTTABLE*.
   i. *MAINACCOUNT*.

[17] As it stands now, the date dimensions have not been normalised into separate dimensions, so it might be reasonable to model date also as a dimension.

6) Finally, the data asset *load_dim_tables_to_dw* upserts the values in the dimension tables. Of note, here the column data types in target Postgres tables correspond to the data types in the source database (MS SQL). Asset *load_dim__fact_rels_to_dw* upserts values to n-m tables (tables denoted by white colour in Figure 9). N-m relational table key columns are based on the concatenation of IDs of linked tables.

The full e-invoice XML fetching pipeline, described earlier, is illustrated in Figure 10, as viewed in the Dagster UI after a successful execution. The Power BI desktop tool generates reports based on the gold-level output of the data pipeline (see Figure 11). The Power BI report directly queries the Postgres DB and deduces the necessary semantic data model for creating and filtering dashboards almost automatically from the source relationships. The only modification required is to enable bi-directional relationships across the automatically inferred semantic data model's relations [84].

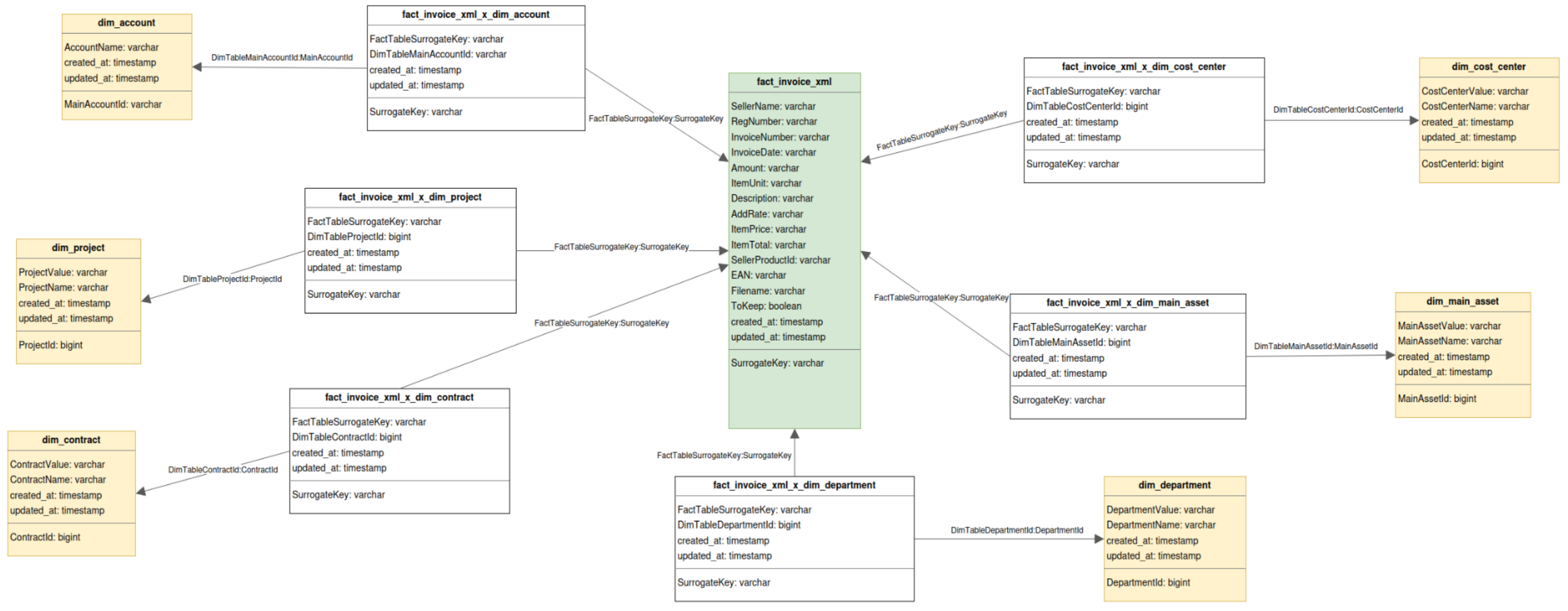


Figure 9. Purchase e-invoice cleaning data pipeline's output to the reporting layer.

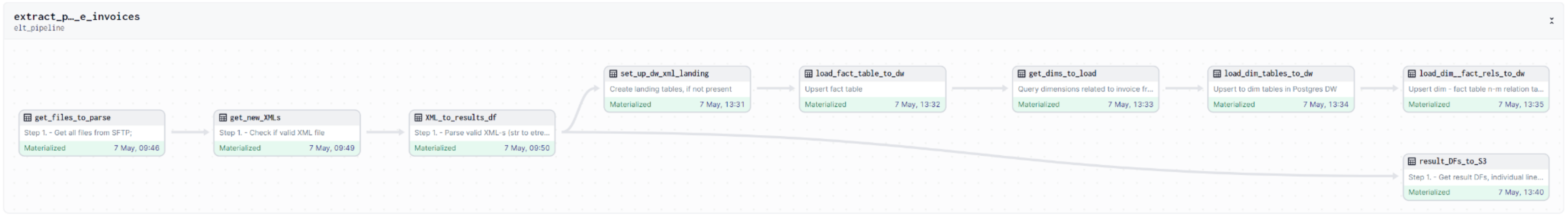
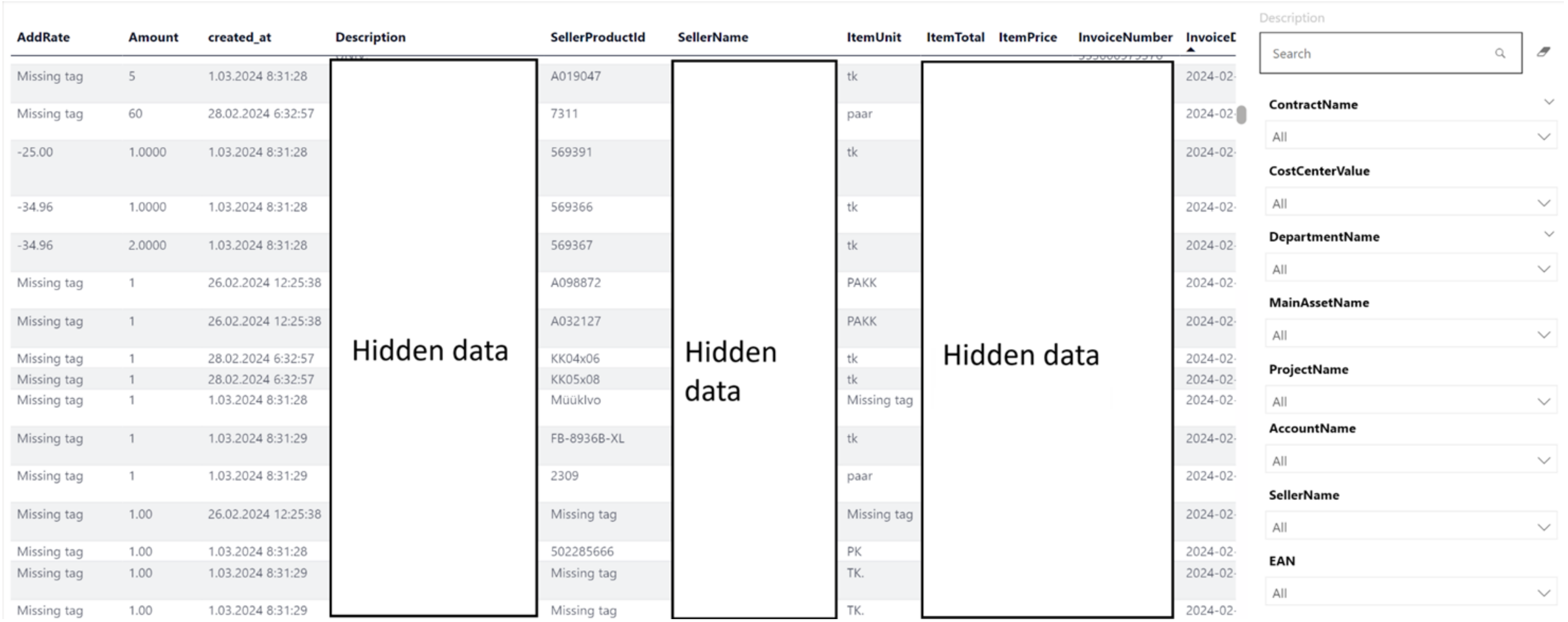

extract_p…_e_invoices
elt_pipeline
get_files_to_parse
Step 1. - Get all files from SFTP;
Materialized 7 May, 09:46
get_new_XMLs
Step 1. - Check if valid XML file
Materialized 7 May, 09:49
XML_to_results_df
Step 1. - Parse valid XML-s (str to etre...
Materialized 7 May, 09:50
set_up_dw_xml_landing
Create landing tables, if not present
Materialized 7 May, 13:31
load_fact_table_to_dw
Upsert fact table
Materialized 7 May, 13:32
get_dims_to_load
Query dimensions related to invoice fr...
Materialized 7 May, 13:33
load_dim_tables_to_dw
Upsert to dim tables in Postgres DW
Materialized 7 May, 13:34
load_dim__fact_rels_to_dw
Upsert dim - fact table n-m relation ta...
Materialized 7 May, 13:35
result_DFs_to_S3
Step 1. - Get result DFs, individual line...
Materialized 7 May, 13:40

Figure 10. Purchase e-invoice XML-s fetching pipeline after successful execution in Dagster UI.



AddRate	Amount	created_at	Description	SellerProductId	SellerName	ItemUnit	ItemTotal	ItemPrice	InvoiceNumber	InvoiceD
Missing tag	5	1.03.2024 8:31:28		A019047		tk				2024-02-
Missing tag	60	28.02.2024 6:32:57		7311		paar				2024-02-
-25.00	1.0000	1.03.2024 8:31:28		569391		tk				2024-02-
-34.96	1.0000	1.03.2024 8:31:28		569366		tk				2024-02-
-34.96	2.0000	1.03.2024 8:31:28		569367		tk				2024-02-
Missing tag	1	26.02.2024 12:25:38		A098872		PAKK				2024-02-
Missing tag	1	26.02.2024 12:25:38		A032127		PAKK				2024-02-
Missing tag	1	28.02.2024 6:32:57	Hidden data	KK04x06	Hidden data	tk	Hidden data			2024-02-
Missing tag	1	28.02.2024 6:32:57		KK05x08		tk				2024-02-
Missing tag	1	1.03.2024 8:31:28		MüükIvo		Missing tag				2024-02-
Missing tag	1	1.03.2024 8:31:29		FB-8936B-XL		tk				2024-02-
Missing tag	1	1.03.2024 8:31:29		2309		paar				2024-02-
Missing tag	1.00	26.02.2024 12:25:38		Missing tag		Missing tag				2024-02-
Missing tag	1.00	1.03.2024 8:31:28		502285666		PK				2024-02-
Missing tag	1.00	1.03.2024 8:31:29		Missing tag		TK.				2024-02-
Missing tag	1.00	1.03.2024 8:31:29		Missing tag		TK.				2024-02-

Description

Search

ContractName
All

CostCenterValue
All

DepartmentName
All

MainAssetName
All

ProjectName
All

AccountName
All

SellerName
All

EAN
All

Figure 11. Purchase e-invoice cleaning data pipeline's output to the reporting layer.

5.3 End-to-end Analysis of Railway Level Crossing Log Data

EVR operates, and will continue to operate in the next 2-3 years, relay-based legacy technology base level crossings. The critical difference between microcontroller-based and relay-based systems is the reduced programmability of the legacy system. Therefore, detecting the approaching train is paramount to ensure secure and reliable operation of the level crossing (i.e., railway crossing with the main road). In detail, if the approaching train detection is lost, typically, the legacy system will initiate the level crossing opening sequence. Microcontroller-based systems would operate under the same hypothetical situation differently; for example, level crossing opening could be postponed if the dime difference between train detection and the release of train detection is spuriously short. By grossly simplifying things, the train's presence is detected by short-circuiting the two electrified rail tracks through the steel wheels of the railway wagon or locomotive undercarriage. Therefore, the train is tracked through discrete blocks, not continually, as the track vacancy is detected through individually wired (term: *rail chains*) track sections (term: *isolation areas*). Due to the steep reduction of cargo train traffic after the Russo-Ukrainian war that would clean the tracks from electricity-isolating dirt, diagnosing train detection losses has become especially important to ensure the safety of railway operations.

EVR utilises the InfraLx information system to manage configuration and collect and visualize data from legacy level crossing systems. This includes detailed, visually presented replays of train passing events that are based on the activation sequence of individual level crossing components. The system has been in operation since 2009 and has seen little to no updates. Atomic event logs from individual subsystems are saved to the Postgres database. Presently, the source system is unable to group atomic log entries into a complete train passing event. In the event of a technical anomaly, the logs are manually checked, cross-referenced with other systems and analysed for diagnostic purposes. It is crucial to note that the InfraLx's source atomic log events are not currently utilised for fully automatic live monitoring and alerting of anomalous events. Understanding the potential benefits of monitoring and alerting, EVR's operations team has requested the following functionality:

1) The system should detect train passing events based on InfraLx's atomic log entries.
2) For every detected train passing event, the duration should be calculated.
3) Visual exploratory data analysis tool that would allow for the depiction of a train passing duration over time by the level crossing, with an emphasised requirement for the possibility to manipulate the temporal zoom level easily.
4) Importantly, to improve monitoring and alerting, the system should allow for setting and updating threshold values for train passing events durations based on lower and upper threshold values. From the operations perspective, train passing events of abnormally short duration have significantly high relevance as those events indicate the possible loss of train detection as the speeding train is, in practice, almost impossible. Furthermore, the criteria values have to be set per level crossing basis as the technical setup differs from level crossing to level crossing quite a bit, e.g. distance from the nearest train stop and positioning of detection points.
5) In the future, i.e. after implementing MS Fabric's data activator functionality [85], the system should send automated MS Teams alerts if the detected train passing event is too short.

The present thesis faces added complexity due to the deployment of InfraLx in a distinct network environment from the staging server. This results in the current live InfraLx being inaccessible from the staging server. To construct the second end-to-end data analysis pipeline, a replica of the VM hosting the InfraLx system was created and hosted in the EVR access network. The analysis undertaken in this thesis pertains to the source data as it existed on January 23rd, 2024, which is the date when the VM hosting the source system's database was copied. During 2023, InfraLx's source system logged 4,709,096 events, roughly corresponding to 12,000 daily log events. The following Figure 12 depicts the E-R schema of the source database.

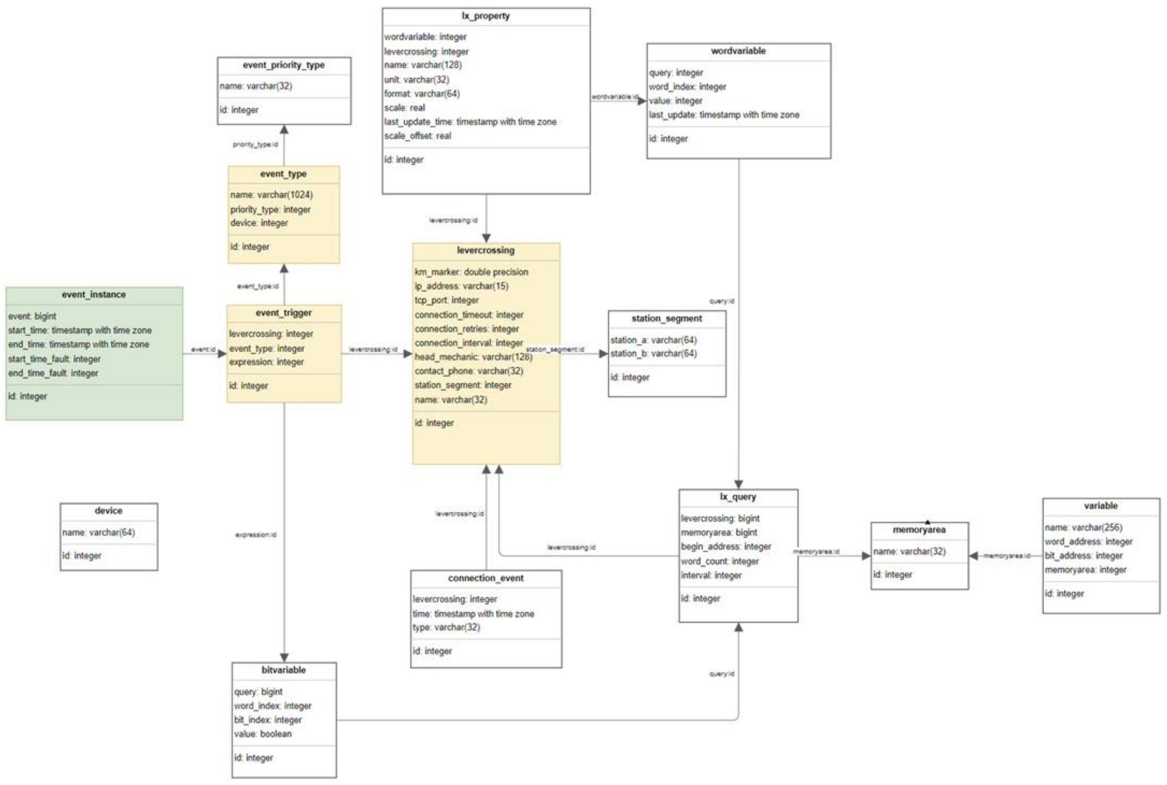


Figure 12. InfraLx source data E-R schema.

For the log data analysis, five new Dagster data assets were created:

1) The data assets *set_up_dw_landing* sets up landing schema, tables, procedures and triggers in the previously referenced Postgres 16-alpine Docker container-based temporary DW. As with the previous purchase e-invoices XML-s pipeline, if required elements are already present in the DW, the step is skipped at the database level. I.e. the Postgres DW I/O Dagster resource calls a Postgres' parameterized table, procedure and trigger creation SQL script templates that delegate the execution logic to DB engine, e.g. skip table creation if the table already exists. In total, two data tables are created:
    a. *source_log_data* - this table is filled with a query from InfraLx DB. The asset fills the target DW staging table with atomic log events and those events' key attributes (event id, level crossing name, event type, start and end datetime, start and end error type).

   b. *matched_log_data* - this table contains the output of train passing matchings. In detail, the table contains the same amount of rows as the table *source_log_data* and the same primary key. Matchings are added to the atomic events, resulting in an output table with columns that describe the source log event id, level crossing name, loge event duration, matched train passing event id (concatenation of source atomic event id-s that make up the train passing event), matched event duration and matched event signature (concatenation of source atomic event types that make up the train passing event).

   Of note, the table for storing per-level crossing criteria values (level crossing name, upper criteria, lower criteria) is created manually.
2) Dagster data asset *load_raw_data* queries data from the source system based on InfraLx's tables that are colour-marked in Figure 12. The only source system's continually updated table is the *event_instance* (marked in green). Other needed source tables (*event_type*, *levelcrossing* and *event_trigger*) for data ingestion query to work (coloured yellow) are in a stale state. As the source table *event_instance* id-s are monotonically growing and InfraLx's log entries are immutable, the data ingestion query uses this value to keep track of already ingested source data, i.e. source data ingestion query is parametrized concerning this value. Ingested data is loaded to DW's table *source_log_data*.
3) Dagster data asset *get_data_to_analyze* loads data for specified level crossings (intra asset parameter) from DW's table *source_log_data* into Python dictionary: keys by level crossing name; values Pandas data frames of source log data.
4) The data asset *match_events* executes the train passing event matching algorithm per level crossing. Three factors complicated the creation of the train-passing detection algorithm:
   a. Firstly, the technical configuration of level crossing systems and thus log event types and train passing event signatures (sequences of source atomic events that correspond to train passing events) vary significantly from level crossing to level crossing.
   b. Secondly, the temporal order of level-crossing events is partially non-deterministic as the analogue relay-based source systems' activation pattern introduces some randomness if the level-crossing system produces log events in a narrow temporal window. For a hypothetical example, if the train passing event encompasses seven atomic events (A, B, C, D, E, F, G) and the first three of those (A-C) happen in narrow time window (< 1s) and the last four events happen in second narrow time window (< 1s) and if we denote the set of all permutations of the first three items {A, B, C} as P(ABC) and the set of all permutations of the last four items {D, E, F, G} as P(DEFG) the valid train passing event signature becomes the union of P(ABC) and P(DEFG).
   c. Thirdly, the atomic event composition of valid train passing events also differs at the intra-level crossing level. For example, a set of atomic elements comprising a valid train passing event can differ in composition, permutation cutoff point(s), and length depending on the direction of the train approach.

The thesis' source repo [69] describes the pattern-matching algorithm in detail through docstrings and in-code comments. Overall, the iterative approach was chosen. For each level crossing, the event signature templates were generated. The candidate event signatures allow for specifying subsets in which template and data sequence comparisons use all the possible permutations from the template. Furthermore, due to previously described circumstances, a single-level crossing might have multiple candidate train passing event signatures, which are then fitted one by one for the source data. Notably, the subsequent iterative passes can only use the source atomic log events that previous pattern-matching iterations have not used. In other words, subsequent pattern matching iterations can only use source log events that form a uniform event sequence over time, i.e. subsequent iterations can match blocks of events whose start and end points do not encompass some atomic log entry already matched with previous matching iteration runs. Evidently, the algorithm currently implemented for train passing event detection is susceptible to the order of the candidate event signatures. As an additional measure to limit false positives, a configurable parameter (with the actually used value of 0.9) was added to the algorithm that discards the event matching if the duration contribution of the first or last atomic pair of train passing event sequence contributes more than parameterized proportion of the whole sequence's duration.

The data asset *load_matched_events_to_dw* upserts matched data to DW's *matched_log_data* table.

Figure 13.0 depicts the previously described level crossing log data analysis pipeline in its entirety, as it is seen in the Dagster UI after a successful execution.

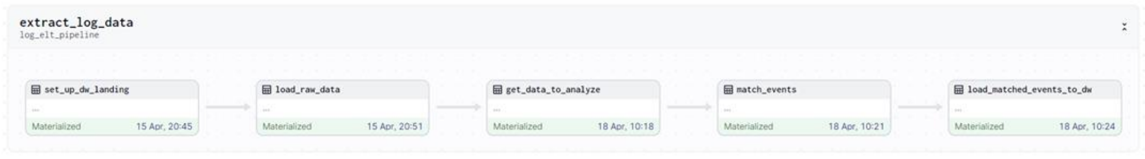
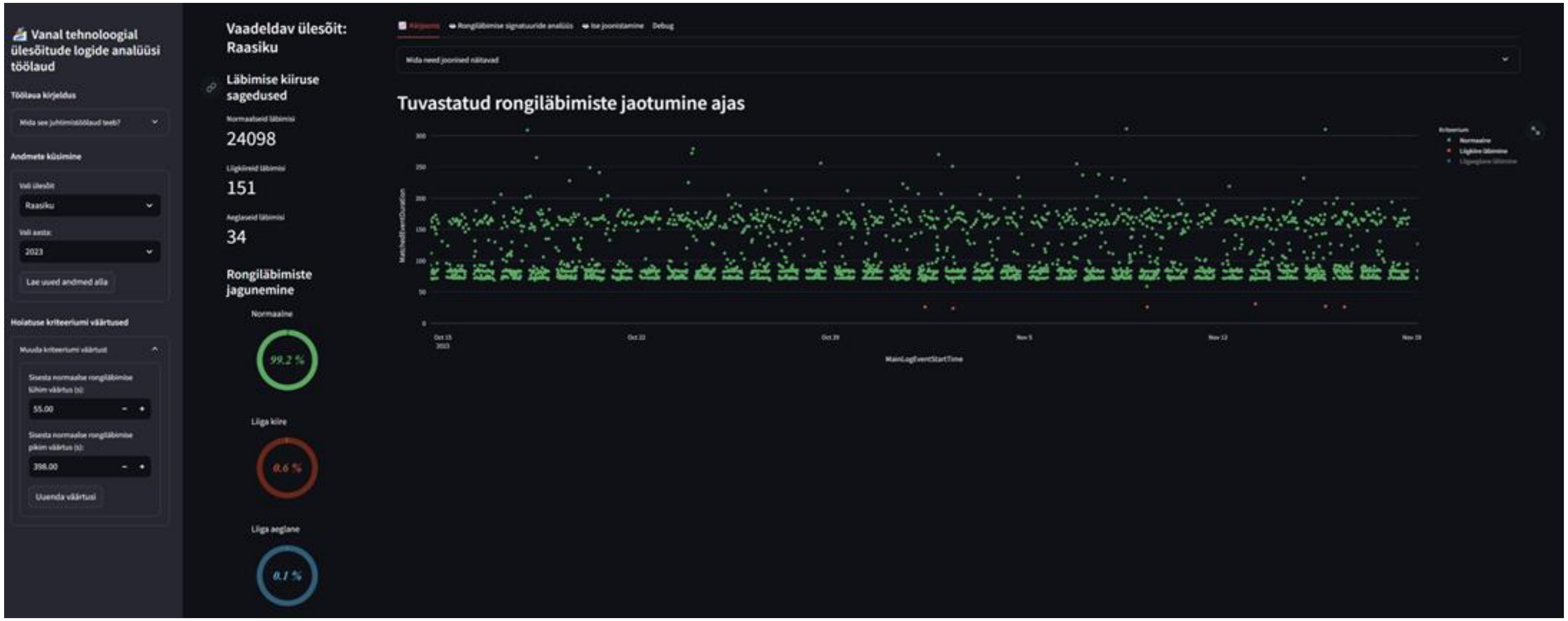


Figure 13. InfraLx source data ER schema.

The reporting based on the data pipeline's output is realised based on the Streamlit [86] dashboard. The dashboard allows for visualising (see Figures 14 - 16) the durations of the matched train passing events by level crossing and year. Notably, the dashboard allows the end user to set criteria values to highlight train passes that were either too short or of too long duration. Criteria values are inserted and updated through a simple CRUD interface and the criteria data is saved to the table in the DW Data is queried from the DW source table *matched_log_data* in order to facilitate the speed of data loading (around 170 000 entries per year per level crossing) downloaded data resource caching [87] and resource (for DB connection) caching [88] is used[18]. The dashboard has the following main elements:

1) Query specification by year and level crossing name.
2) Criteria values update CRUD fields.
3) Quick summary statistics pane that is computed based on selections and values from the last two sections (see left side of the Figures' 14-16 black-coloured area).
4) Tab for displaying train passing events timings over the selected year and level crossing (see Figure 14).
5) Tab for displaying train passing events timings over the selected year, matched event signature and level crossing (see Figure 15).
6) *PyGWalker* [89] based self-service environment for exploratory data analysis (see Figure 16).

[18] By default, Streamlit's execution model executes all the report's code from start to finish after every input change [90]. Furthermore, Streamlit lacks an explicit way to control asynchronous code execution order. Therefore, to avoid re-running long-running data loading steps, i.e. direct queries against the Postgres DW, Streamlit suggests using data and resource caching.


Vanal tehnoloogial ülesõitude logide analüüsi töölaud

Töölaua kirjeldus

Mida see juhtimistöölaud teeb?

Andmete küsimine

Vali ülesõit: Raasiku

Vali aasta: 2023

Lae uued andmed alla

Hoiatuse kriteeriumi väärtused

Muuda kriteeriumi väärtust

Sisesta normaalse rongläbimise lühim väärtus (s): 55.00

Sisesta normaalse rongläbimise pikim väärtus (s): 398.00

Uuenda väärtusi

Vaadeldav ülesõit: Raasiku

Läbimise kiiruse sagedused

Normaalseid läbimisi: 24098

Liigkiireid läbimisi: 151

Aeglaseid läbimisi: 34

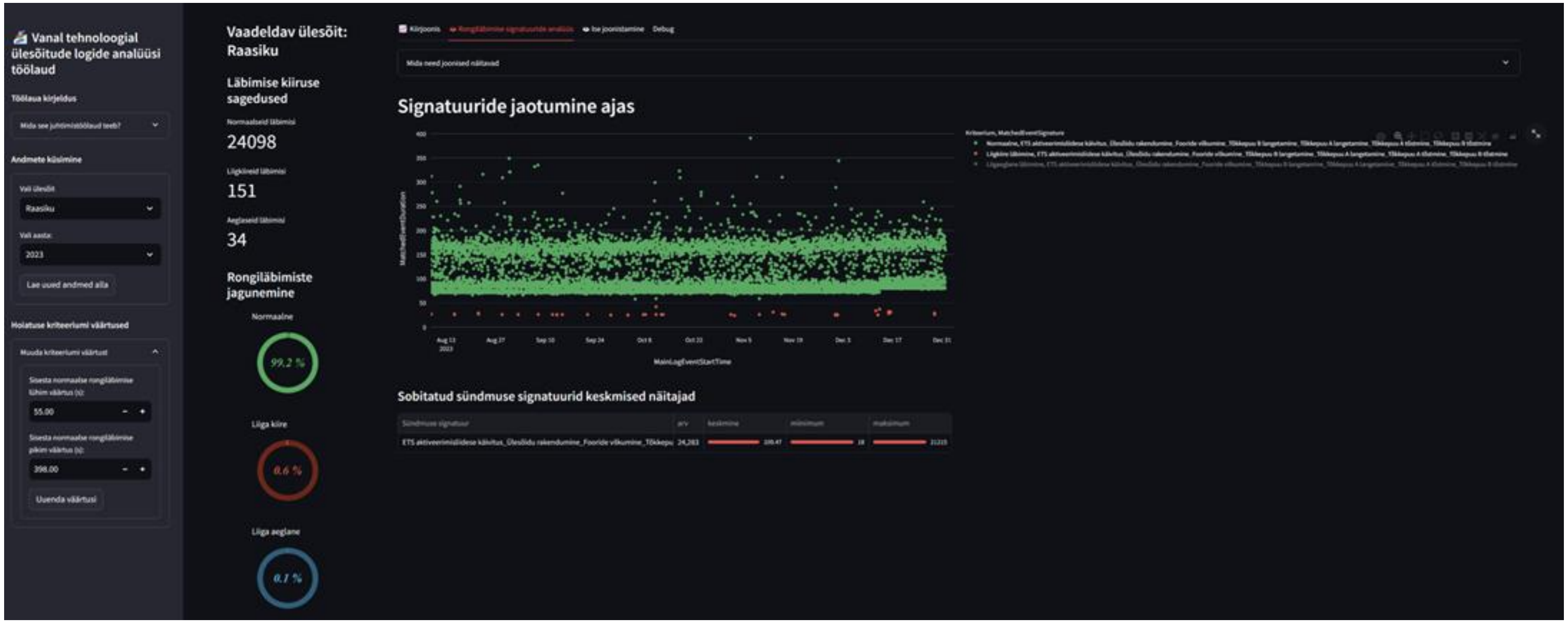
Rongiläbimiste jagunemine

Normaalne: 99.2 %

Liiga kiire: 0.6 %

Liiga aeglane: 0.1 %

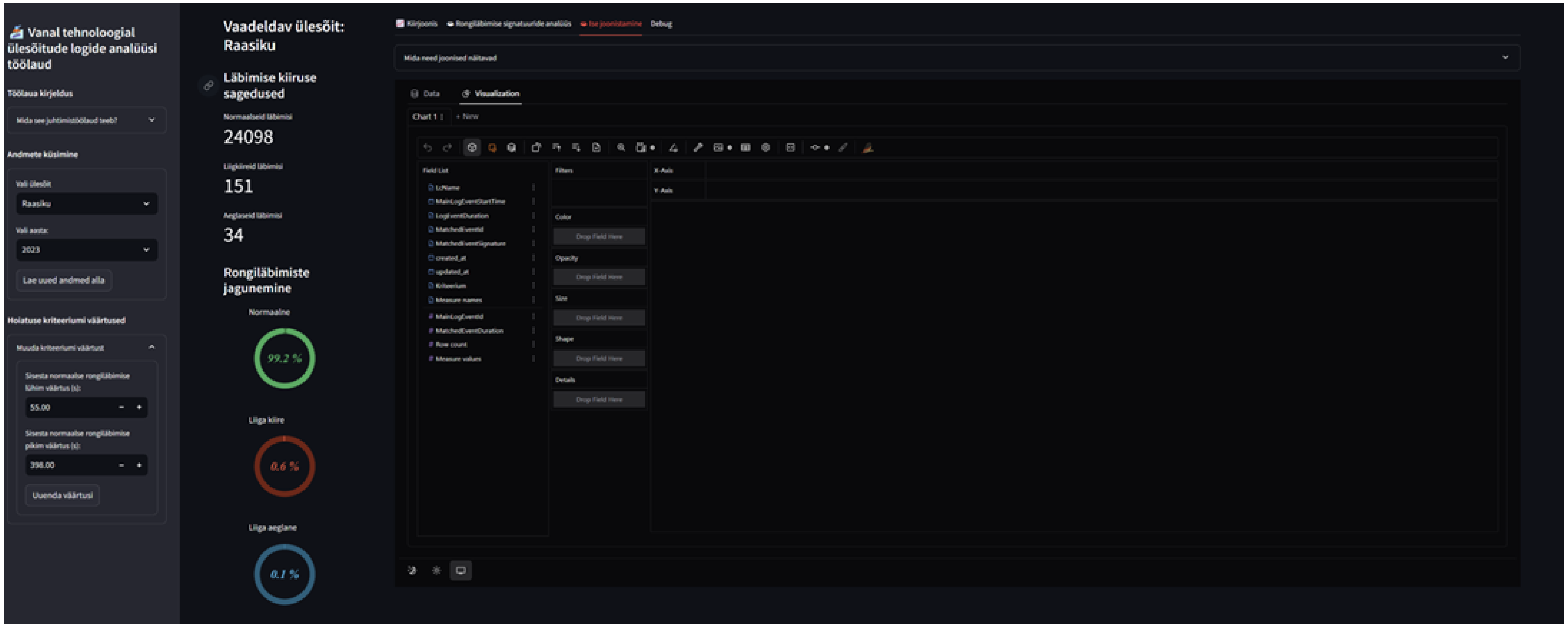
Figure 14. Level crossing log data analysis dashboard - matched event durations.


Vanal tehnoloogial ülesõitude logide analüüsi töölaud
Töölaua kirjeldus
Mida see juhtimistöölaud teeb?
Andmete küsimine
Vali ülesõit
Raasiku
Vali aasta:
2023
Lae uued andmed alla
Hoiatuse kriteeriumi väärtused
Muuda kriteeriumi väärtust
Sisesta normaalse rongläbimise lühim väärtus (s):
55.00
Sisesta normaalse rongläbimise pikim väärtus (s):
398.00
Uuenda väärtusi
Vaadeldav ülesõit: Raasiku
Läbimise kiiruse sagedused
Normaalseid läbimisi
24098
Liigkiireid läbimisi
151
Aeglaseid läbimisi
34
Rongiläbimiste jagunemine
Normaalne
99.2 %
Liiga kiire
0.6 %
Liiga aeglane
0.1 %
Kiirjoonis
Rongiläbimise signatuuride analüüs
Ise joonistamine
Debug
Mida need joonised näitavad
Signatuuride jaotumine ajas
400
350
300
250
200
150
100
50
0
MatchedEventDuration
Aug 13 2023
Aug 27
Sep 10
Sep 24
Oct 8
Oct 22
Nov 5
Nov 19
Dec 3
Dec 17
Dec 31
MainLogEventStartTime
Kriteerium, MatchedEventSignature
Normaalne, ETS aktiveerimisliidese käivitus, Ülesõidu rakendumine_Fooride vilkumine_Tõkkepuu B langetamine_Tõkkepuu A langetamine_Tõkkepuu A tõstmine_Tõkkepuu B tõstmine
Liigkiire läbimine, ETS aktiveerimisliidese käivitus, Ülesõidu rakendumine_Fooride vilkumine_Tõkkepuu B langetamine_Tõkkepuu A langetamine_Tõkkepuu A tõstmine_Tõkkepuu B tõstmine
Liigaeglane läbimine, ETS aktiveerimisliidese käivitus, Ülesõidu rakendumine_Fooride vilkumine_Tõkkepuu B langetamine_Tõkkepuu A langetamine_Tõkkepuu A tõstmine_Tõkkepuu B tõstmine

Sobitatud sündmuse signatuurid keskmised näitajad

Sündmuse signatuur	arv	keskmine	minimum	maksimum
ETS aktiveerimisliidese käivitus_Ülesõidu rakendumine_Fooride vilkumine_Tõkkepu	24,283	109.47	18	21215

Figure 15. Level crossing log data analysis dashboard - analysis by event signatures.



Vali ülesõit: Raasiku
Vali aasta: 2023
Sisesta normaalse rongiläbimise lühim väärtus (s): 55.00
Sisesta normaalse rongiläbimise pikim väärtus (s): 398.00

Normaalseid läbimisi: 24098
Liigkiireid läbimisi: 151
Aeglaseid läbimisi: 34

Normaalne: 99.2 %
Liiga kiire: 0.6 %
Liiga aeglane: 0.1 %

Figure 16. Level crossing log data analysis dashboard – exploratory data analysis view.

6 Conclusions

The thesis was written with the aim of aiding a railway company in analysing and developing a strategic plan for enhancing its analytics data management platform. A key recommendation from the thesis is the adoption of a flexible hybrid architecture that integrates on-premises/Infrastructure as a Service (IaaS) with Software as a Service (SaaS) solutions. This approach is customised to accommodate the diverse complexity and types of data sources within the company, suggesting a tailored solution rather than a one-size-fits-all model.

Moreover, the thesis successfully tested the essential elements of the proposed architecture by addressing two real-world usage cases: analysis of purchase invoices and railway level crossings' log data. Consequently, the thesis illustrates how the suggested data management architecture can notably improve EVR's data engineering capabilities.

Moving forward, EVR has many chances to continue improving its platform for analysing data. The thesis didn't cover one promising improvement: using parallel computing solutions, like DASK with the Modin package, in the TO-BE architecture's on-premises or Infrastructure as a Service (IaaS) part. This could make data processing faster and more scalable. By adopting such advanced technologies, EVR can further enhance its ability to process data.

7 Acknowledgments

I want to express my deepest gratitude to my supervisors, Kristo Raun and Prof. Ahmed Awad. Their valuable insights, support, and consistent guidance have been invaluable during the thesis writing period. I appreciate their honest feedback and the wealth of technical ideas and suggestions they shared. From the student's perspective, this has been the most supported thesis writing and supervision experience thus far.

I want to take a moment to express my heartfelt thanks to my fantastic wife, Conghui, for her support and understanding while I was working on my thesis. I especially thank her and her family for three weeks in lovely Yunnan[19] province, which allowed me to get three weeks of breathing space away from a super intensive work environment. While staying in China, I completed bulk of the thesis, background reading, and a large part of the log analysis pipeline.

I am further expressing my gratitude to Aare for carefully reviewing my thesis and providing invaluable hinting about sections that are grinding to read. A huge thanks to Aivar Mihhailov from EVR for inspiring the idea for log analysis and to my colleague Raini for helping me untangle the complexities of the Dynamics D365 database and for proposing the XML-s as worthy data source to try out. I also want to thank my supervisor Mailis and colleagues Denis and Jüri for their assistance in ensuring the staging server computing resource and for their pivotal role in getting the InfraLx database copy VM up and running. I thank our IT architect Aleksandr and IT analyst Külli Kivi for explaining the intricacies of VJS and PONY. Last but certainly not least, I want to express my sincere appreciation to our network engineer, Ain, and the head of the department, Tõnu, for their support in navigating the deployment's security considerations. I couldn't have done it without all of your help!

7.1 Methodological Acknowledgments

The author acknowledges using Grammarly, a recommended LLM-based tool by the Institute of Computer Science, as a writing coach to maintain a score of approximately 92-95 (accessed via the university's licence). Of note is that the author initially wrote the text predominantly in Google Docs, and Grammarly's utilization followed. This initial material from Google Docs included rephrased and synthesized text from referenced sources. No references have been sourced from LLM tools nor content *de-novo* synthesized by LLM tools. LLM tool usage involved following Grammarly's spellcheck, wording, and other automatic suggestions to improve the text's readability score. This less intrusive assistance encompasses around 50% of the text. Additionally, the author utilized Grammarly's generative AI capabilities to help refine the text at a more involved level. Working problematic paragraph by problematic paragraph, Grammarly was instructed to rework the select paragraph with the following custom prompts: "Make it sound academic", "Make it more concise", "Lessen the usage of passive voice", "Summarize academically the main point of the text", "Summarize the main point of the text in one/two sentences" and "Make the text shorter and easier to read". The more assisted mode of Grammarly usage covers the remaining half of the thesis. As an example of less intrusive Grammarly usage, the author wrote

[19] Objectively the second-best province in China after Heilongjiang province and its snowy capital Harbin, the city which also exists in one of the tables of VJS and from which you can send goods via railway to Estonia with SMGS rail consignment note that the VJS system can also understand.

this rather lengthy paragraph himself but accepted Grammarly's automatic suggestions during the writing process[20].

The practical implementation's codebase and docstrings were not generated by any LLM or Gen AI tool. The coding work was supported by utilizing code examples in the tool's documentation, official tool example repos, and tutorial repos. These resources provided the carcass for developing EVR-specific solutions.

[20] As for examples of the author's level of academic writing before the advent of LLM-based assistants, see work from 2016 https://www.sciencedirect.com/science/article/abs/pii/S0924977X16000651 and work from 2018 https://dspace.ut.ee/items/11c0950b-dcb6-4bb6-978b-27e8382f1e87. This paragraph has Grammarly’s score of 98.

References

[1] Volk, M.; Staegemann, D.; Pohl, M.; Turowski, K. "Challenging Big Data Engineering: Positioning of Current and Future Development." In IoTBDS, pp. 351-358. 2019.

[2] Reis, J.; Housley, M. Fundamentals of Data Engineering. " O'Reilly Media, Inc.", 2022.

[3] Smitis, A.; Skiadopoulos, S.; Vassiliadis, P. "The History, Present, and Future of ETL Technology." In DOLAP, pp. 3-12. 2023.

[4] MS Fabric Introductory Slide Decks. https://github.com/microsoft/Fabric-Readiness

[5] Patel, M.; Patel, D. B. "Progressive growth of ETL tools: A literature review of past to equip future." Rising Threats in Expert Applications and Solutions: Proceedings of FICR-TEAS 2020 (2020): 389-398.

[6] What is Microsoft Fabric? https://learn.microsoft.com/en-us/fabric/get-started/microsoft-fabric-overview

[7] Cloudera Data Platform (CDP). https://www.cloudera.com/products/cloudera-data-platform.html

[8] Databricks platform description. https://www.databricks.com/product/data-intelligence-platform

[9] What is Amazon EMR? https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html

[10] What is AWS Glue? https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html

[11] What is Amazon Redshift? https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html

[12] Dash, B.; Swayamsiddha, S. "Reverse ETL for Improved Scalability, Observability, and Performance of Modern Operational Analytics - a Comparative Review." In 2022 OITS International Conference on Information Technology (OCIT), pp. 491-494. IEEE, 2022.

[13] Tutcher, J. "Ontology-driven data integration for railway asset monitoring applications." In 2014 IEEE International Conference on Big Data (Big Data), pp. 85-95. IEEE, 2014.

[14] Davari, N.; Veloso, B.; Costa G. A. Pedro Mota Pereira, Rita P. Ribeiro, and João Gama. "A survey on data-driven predictive maintenance for the railway industry." Sensors 21, no. 17 (2021): 5739.

[15] Ghofrani, F.; He Q.; Goverde R.; Liu, X. "Recent applications of big data analytics in railway transportation systems: A survey." Transportation Research Part C: Emerging Technologies 90 (2018): 226-246.

[16] Binder, M,; Mezhuyev, V.; Tschandl, M. "Predictive maintenance for railway domain: A systematic literature review." IEEE Engineering Management Review 51, no. 2 (2023): 120-140.

[17] Armbrust, M.; Ghodsi, A.; Xin, R.; Zaharia, M. "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics." In Proceedings of CIDR, vol. 8, p. 28. 2021.

[18] Singhal, B.; Aggarwal, A. "ETL, ELT and reverse ETL: a business case Study." In 2022 Second International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE), pp. 1-4. IEEE, 2022.

[19] Bennie, H.; Davis, D. Delta Lake: Up and Running. " O'Reilly Media, Inc.", 2023.

[20] Wiak, S.; Drzymala, P.; Welfleprzeglad, P. "Using ORACLE tools to generate Multidimensional Model in Warehouse." ELEKTROTECHNICZNY (Electrical Review), ISSN (2012): 0033-2097.

[21] Databricks website: What is a medallion architecture? https://www.databricks.com/glossary/medallion-architecture

[22] Spark project website: Unified engine for large-scale data analytics. https://spark.apache.org/

[23] Mahmoud, A. S. I. Data Warehouse Modelling Using Data Vault 2.0 in Fintech Companies - Alumni Talks 2023. https://www.youtube.com/watch?v=02eCishUY10&t=1327s

[24] Khononov, V. Learning Domain-Driven Design. " O'Reilly Media, Inc.", 2021.

[25] Vernon, V. Domain-driven design distilled. Addison-Wesley Professional, 2016.

[26] Dehghani, Z. Data Mesh. Marcombo, 2022.

[27] Machado, I.A.; Costa, C.; Santos, M.Y. "Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures." Procedia Computer Science 196 (2022): 263-271.

[28] Evans, E. Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2004.

[29] Kimball, R; Ross; M. The data warehouse toolkit: the complete guide to dimensional modeling 2nd edition. John Wiley & Sons, 2002.

[30] Azure documentation: What is Azure VPN Gateway? https://learn.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways

[31] Metadata Management: Benefits, Automation, Use cases, and Framework. https://atlan.com/metadata-management-101/#what-is-metadata-management

[32] Ereth, J. "DataOps-Towards a Definition." LWDA 2191 (2018): 104-112.

[33] Apache Doris - Open Source, Real-Time Data Warehouse. https://doris.apache.org/

[34] TiDB introduction on BingCap’s webpage. https://www.pingcap.com/tidb/

[35] Oracle DB documentation: In-Memory Column Store Architecture. https://docs.oracle.com/en/database/oracle/oracle-database/21/inmem/in-memory-column-store-architecture.html#GUID-D61E56A9-B152-49D1-9956-BE9E882E3DE1

[36] MS SQL documentation: Management data warehouse. https://learn.microsoft.com/en-us/sql/relational-databases/data-collection/management-data-warehouse?view=sql-server-ver16

[37] MS SQL documentation: Columnstore indexes - Data Warehouse. https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-data-warehouse?view=sql-server-ver16

[38] Federated queries in data lakes with Redpanda and Trino. https://redpanda.com/blog/data-lake-query-federation-tutorial

[39] Trino documentation: available source connections. https://trino.io/docs/current/connector.html

[40] Raghav, S.; Traverso, M.; Sundstrom, D.; Phillips, D.; Xie, W.; Sun, Y.; Yegitbasi, N. "Presto: SQL on everything." In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1802-1813. IEEE, 2019.

[41] DuckBB website: DuckDB, the great federator? https://motherduck.com/blog/duckdb-the-great-federator/

[42] Mukherjee, R.; Kar, P. "A comparative review of data warehousing ETL tools with new trends and industry insight." In 2017 IEEE 7th International Advance Computing Conference (IACC), pp. 943-948. IEEE, 2017.

[43] Apache Druid project webpage: Introduction to Apache Druid. https://druid.apache.org/docs//0.22.0/design/index.html

[44] Apache Pinot project website: What is Apache Pinot? https://github.com/apache/pinot?tab=readme-ov-file

[45] Clickhouse company homepage. https://clickhouse.com/

[46] DASK project website. https://www.dask.org/

[47] Teradata website's transactional workloads description. https://www.teradata.com/platform/workloads/transactional

[48] Buuck, B. Minio blog: Unbundling the Data Stack: the Disaggregation of Storage and Compute 2.0. https://blog.min.io/disaggregation-of-storage-and-compute-2-0/

[49] Buuck, B. Minio blog: Databases for an Object Storage Centric World. https://blog.min.io/databases-for-object-storage/

[50] Apache Iceberg project website: Apache Iceberg Quickstart https://iceberg.apache.org/spark-quickstart/

[51] Apache Hudi project website: What is Apache Hudi. https://hudi.apache.org/

[52] Delta Table webpage: Build Lakehouses with Delta Lake. https://delta.io/

[53] Example GitHub repo to exemplify dockerising Hive-metastore. https://github.com/naushadh/hive-metastore

[54] MinIO S3 documentation: MinIO Object Storage for Container. https://min.io/docs/minio/container/index.html

[55] Li, C.; Lodin, J. Trino Community Broadcast 34: A Big Delta for Trino. https://trino.io/episodes/34.html

[56] Dagster documentation: Deploying Dagster to Docker. https://docs.dagster.io/deployment/guides/docker

[57] MS Fabric documentation: Work with Delta Lake tables in Microsoft Fabric. https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/

[58] Informatica website: Informatica Integration Hub. https://www.informatica.com/gb/products/data-integration/integration-hub.html

[59] Matillion website: How the Matillion ETL tool works. https://www.matillion.com/matillion-etl

[60] Dagster documentation: Deploying Dagster on Helm. https://docs.dagster.io/deployment/guides/kubernetes/deploying-with-helm

[61] Airbyte: UI Overview. https://www.restack.io/docs/airbyte-knowledge-airbyte-ui-overview

[62] Pimentel, J. F.; Murta L.; Braganholo, V.; Freire, J. "A large-scale study about quality and reproducibility of jupyter notebooks." In 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp. 507-517. IEEE, 2019.

[63] Pimentel, J. F.; Murta L.; Braganholo, V.; Freire, J. "Understanding and improving the quality and reproducibility of Jupyter notebooks." Empirical Software Engineering 26, no. 4 (2021): 65.

[64] What is a Data Contract? https://www.datamesh-manager.com/learn/what-is-a-data-contract

[65] MS Fabric documentation: Fabric domains. https://learn.microsoft.com/en-us/fabric/governance/domains

[66] Tsafir, N. Easily implement data mesh architecture with domains in Fabric https://blog.fabric.microsoft.com/en-us/blog/easily-implement-data-mesh-architecture-with-domains-in-fabric/

[67] Airbyte documentation: Getting Started with PyAirbyte (Beta). https://docs.airbyte.com/using-airbyte/pyairbyte/getting-started

[68] OpenMetadata website. https://open-metadata.org/

[69] Thesis GitHub repo. https://github.com/Mait22/DS-MSc-AY2024

[70] Oracle Linux. https://www.oracle.com/linux/

[71] Docker documentation: Install Docker Engine on CentOS. https://docs.docker.com/engine/install/centos/

[72] Docker documentation: Install the Compose plugin https://docs.docker.com/compose/install/linux/

[73] Rahandusministeeriumi koduleht: E-arved. https://www.fin.ee/riigi-rahandus-ja-maksud/riigi-raamatupidamine/e-arved

[74] Masintöödeldava algdokumendi juhendi kehtestamine. https://www.riigitea-taja.ee/akt/113042017005

[75] Description of Estonian e-invoice. https://billberry.ee/help/dev/estonian-einvoice-standard/estonian-einvoice-standard-1.2.EN.en.pdf

[76] Github: atmoz/sftp. https://github.com/atmoz/sftp

[77] Dagster documentation: Resources. https://docs.dagster.io/concepts/resources

[78] Dagster documentation: I/O managers. https://docs.dagster.io/concepts/io-mana-gement/io-managers

[79] PyPI: pysftp. https://pypi.org/project/pysftp/

[80] PyPI: minio. https://pypi.org/project/minio/

[81] PyPI: lxml. https://pypi.org/project/lxml/

[82] Dagster documentation: Software-defined assets. https://docs.dagster.io/concepts/as-sets/software-defined-assets

[83] MS D365 documentation: Dimension code combination in Main(DimensionAttribu-teValueCombination. https://learn.microsoft.com/en-us/common-data-mo-del/schema/core/operationscommon/tables/finance/financialdimensions/main/dimen-sionattributevaluecombination

[84] Power BI documentation: Enable bidirectional cross-filtering for DirectQuery in Po-wer BI Desktop. https://learn.microsoft.com/en-us/power-bi/transform-model/desk-top-bidirectional-filtering

[85] MS Fabric Documentation: What is Data Activator? https://learn.microsoft.com/en-us/fabric/data-activator/data-activator-introduction

[86] Get started with Streamlit. https://docs.streamlit.io/get-started

[87] Streamlit documentation: st.cache_data. https://docs.streamlit.io/develop/api-refe-rence/caching-and-state/st.cache_data

[88] Streamlit documentation: st.cache_resource. https://docs.streamlit.io/develop/api-re-ference/caching-and-state/st.cache_resource

[89] PyGWalker. https://github.com/Kanaries/pygwalker

[90] Working with Streamlit's execution model. https://docs.streamlit.io/develop/con-cepts/architecture

Appendix

I. Glossary of Abbreviations

- ACID – an acronym that stands for Atomicity, Consistency, Isolation, and Durability.
- API – application programming interface.
- CCS – common control system.
- CDC – changed data capture.
- CRUD– create, read, update, delete data operations.
- CSZT – Council for Railway Transport of the Commonwealth Member States
- D365 – Microsoft Dynamics 365 enterprise resource planning system.
- DB – database.
- DNS – domain name service.
- DW – data warehouse.
- ELT – extract, load, transform.
- ETL – extract, transform, load.
- ERP – enterprise resource planning system.
- EVR DW – EVR's data warehouse.
- gRPC– remote procedure call framework.
- ICF – next generation level crossing system manufacturer.
- KPI – key performance indicator.
- OLAP – online analytical processing.
- OLTP – online Transaction Processing.
- PBI – Microsoft Power BI reporting tool.
- PL/SQL – Oracle Corporation's procedural extension for SQL.
- POC – proof of concept.
- PONY – EVR's internal information system that is built over time to support a diverse and loosely coupled set of business functions (incident management, hazard assessment and management, management of infrastructure master etc).
- SLA – service-level agreement.
- SP – SharePoint.
- SSIS – SQL Server Integration Services.
- TCP – Transmission Control Protocol.
- TTCMS – in-development future traffic management system.
- UI – user interface.
- VJS – EVR's wagon management system (in Estonian: Vedude Juhtimise Süsteem/Vagunite Jälgimise Süsteem).
- VLAN – virtual local area network.
- VM – virtual machine.
- WD – Web Desktop, document management software.

II. TO-BE Architecture and IT-Business Cooperation Model

Figure 7 illustrates the TO-BE architecture, highlighting the distinct boundaries of responsibility between IT and business based on the selected analytic data management model. In the case of a centralised approach (refer to decision dimension no. 1 in section 4.1), the IT department becomes the *de facto* owner of the analytic data. Specifically, the data lake/data warehouse development team would need to represent the analytic data requirements during the requirements engineering phase of the source system, establish corresponding data ingestion and analytic domain objects extraction pipelines, and ensure the proper storage of the gold layer analytic data across domains, thus ensuring the cross-domain consistency of master data. In a centralised approach, the IT department must oversee the self-serve endpoints for central data access, as the technical department holds the overall picture of stored gold layer data. The primary source of consuming analytic data for the business side would remain centrally developed Power BI reports. It is evident that if the centralised approach were adopted, much of the Fabric platform's data mesh-specific functionalities would go unused.

The diagram in Figure 17 outlines a proposed model for managing analytical data that can fully leverage the potential of the TO-BE architecture. The challenge of adopting data mesh-based thinking is similar to implementing a microservices-based system architecture. This involves breaking down the monolithic system or data into meaningful autonomous parts that align with domain-specific business requirements and rules.

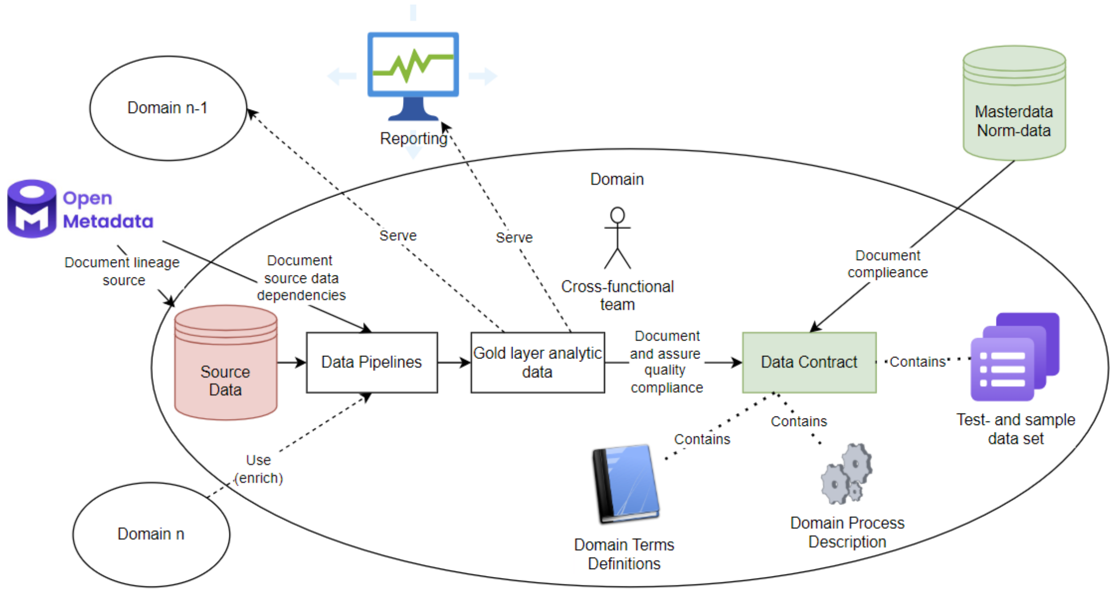


Figure 17. Outline of the data mesh enforcement under TO-BE analytic data management architecture.

In practice, implementation of data mesh means working towards creating and utilising a shared language, known as the ubiquitous language, which bridges the gap between domain experts, self-serve data users, and developers. Additionally, at a strategic level, data implementation entails defining bounded contexts and establishing explicit boundaries within which a specific model is applicable. At the level of analytic data, the ubiquitous language is explicated in the data contract, which describes the main domain-specific terms and processes needed to use the analytic data on a self-serve basis or as an enrichment input in another domains' data pipelines; this is in addition to stating quality standards and service-level agreements (SLAs) associated with the given published dataset.

III. Technology Alternatives for Practical Use Cases

When selecting the middle layer technologies for TO-BE's architecture, the primary consideration is the ability to experiment freely at no cost, with the potential to deploy on an IaaS or PaaS/SaaS basis in the future. These criteria have led to limited technology options for the middle layer, consisting solely of open-source projects backed by mid to large-sized technology companies.

The most obvious question arising about TO-BE technological stack is the choice of Dagster over the industry standard orchestration tool Airflow. As outlined above, the problems inherent in the current technical setup are a need for containerization, code maintainability, and the possibility of running, debugging, and developing the code locally independently of the production system. Those areas needing improvement are also why currently deploying and running multiple versions of the same end-to-end data pipeline is nearly impossible to accomplish. Across all the above-outlined development needs, Dagster has clear advantages over Airflow:

1) Firstly, Dagster provides robust local development, unit testing, and staging environment support. This feature set allows developers to work with pipelines outside production deployments, making testing, developing, and debugging pipelines easier. On the other hand, Airflow often requires a production context to be workable.
2) Additionally, Dagster is designed for container-native orchestration, making it ideal for modern hybrid or cloud-native environments. Conversely, containerizing and deploying the Airflow setup on the Kubernetes cluster requires a more complex manual setup.
3) Finally, while Airflow primarily focuses on orchestration, Dagster prioritises data assets and modular transformation based on those assets. Dagster is inherently aware of data passing between pipeline steps and includes built-in pickle support for basic Python data types (lists, tuples, dictionaries) and commonly used Python data tools like Pandas and Polars.

Dagster and Airflow offer a convenient UI-based admin view with scheduling and pipeline execution health monitoring. However, Dagster's drawback lies in the need for built-in authentication capacity. Moreover, due to Dagster's daemon node's reliance on GraphQL-based API endpoints to communicate with the UI node, implementing basic authentication via a reverse proxy setup can be complex.

As for alternative deployment paths, there is a promising opportunity to transition the TO-BE setup to a cloud-native deployment model. This would entail:

1) The current on-prem Minio S3 installation can be replaced with Amazon S3, Azure's Data Lake Gen 2 object storage, or Fabric's own OneLake storage.
2) Similarly, the on-prem relational Postgres storage can be substituted with Azure's, Oracle's Cloud, or Amazon's Postgres-managed instance offerings. Alternatively, Fabric's built-in DataWarehouse instance and CRUD operations with the Synapse Data Warehouse SQL engine via JCDB-endpoint are also feasible options.
3) OpenMetadata provides a convenient SaaS deployment option that is fully managed.
4) Additionally, Airbyte can also be accessed as a fully managed SaaS option when required. This enables the extraction of data from external system API endpoints or the running of CDC outside of Fabric's platform.
5) Finally, Dagster presents a deployment option in which the daemon and UI nodes are deployed on the Dagster cloud, while the worker nodes are located on-premises or on an IaaS basis on the Amazon AWS platform.

For the sake of completeness, it's important to note that several commercial data movement and pipeline orchestration tools exist that, if combined with an adequate storage layer, could approximate the functionality of the architecture advanced by the current thesis. Numerous industrial and infrastructure management companies throughout the Baltic and Nordic regions seem to utilise alternative analytic tech stacks. This conclusion is based on a non-scholarly assessment of the author's LinkedIn and CV.ee adverts about relevant job postings. Examples of comparable technical options include:

1) Matillion for data extraction and transformation, coupled with Snowflake analytic storage and Tableau-based reporting. Matillion provides a cloud-native data integration platform, enabling users to perform complex data integration, transformation, and ETL operations in a low-code/no-code environment. Meanwhile, Snowflake is a cloud-based data warehousing platform that separates computing and storage resources, allowing users to pay solely for the aids they require, and it employs SQL as its query language.
2) Another option that seems to be commonly utilised, particularly in the energetics and IoT fields, is a comprehensive solution that leverages Hitachi Vantara's product portfolio. This approach involves using Pentaho for purposes similar to Matillion while relying on traditional relational data warehousing and object storage for storage needs.
3) Alternatively, some companies seem to have chosen to utilise on-premises MS SQL or Oracle-based relational data warehousing, combined with T-SQL or PL/SQL procedures for data transformations, along with Power BI-based reporting.

IV. Cost-capacity Analysis of Microsoft Fabric Data Lakehouse SaaS Platform

Regarding costs, Microsoft Fabric's F64 ability tier, which has 64 capacity units (CU) and 32GB of RAM, costs 5000 euros monthly. The F64 tier is the cheapest option for deploying Microsoft Fabric against Azure's vNet infrastructure (allowing for secure private IP-based networking between cloud and on-prem resources over a VPN channel). Further, as a bonus, the F64 tier offers unlimited Power BI end-user licences, provided that concurrent Power BI end-user sessions have to be serviceable with underlying resources of 8 CUs and 4 GB of RAM. It is essential to emphasise Microsoft's accounting for offered computing capacity, with one CU equaling the computing power that would yield 2000 points in the CoreMark benchmark suite. Taking AMD's ZEN 5 microarchitecture-based Epych 32-core processor as a reference roughly yields one CU equaling 1/8 of one real CPU core. 5000 Euros for a core single node machine with 32 GB of RAM (22 GB available for compute use) and no GPU resources is 10-20 times the markup compared to infrastructure deployed on-prem or on an IaaS basis.

Engaging in hypothetical discussions about pricing is crucial as they have a significant impact on the future. If EVR's analytic demands surpass the computing capacity of a single eight-core VM, upgrading to Fabric's next performance tier (128 CU, 16 core, 64 GB of memory) comes at a cost of approximately 5000 €. This amount is comparable to the salary budget of a full-time DataOps specialist.

Furthermore, considering that Apache Spark as technology focuses on solving the problem of handling and transforming large datasets that do not fit in a single compute node's RAM (i.e. the requirement for distributed compute), running Spark cluster on top of Fabric's F64 instance that is by today's standard low to medium performing single node compute equivalent, is a bit of an oxymoron. Such deployment introduces overhead both in the compute (coordinator node in addition to worker nodes) and complexity (need to follow Spark's API), which is unnecessary as programmatically more straightforward and on a single CPU, highly parallelized options like Polars and Pandas drop-in replacement Modin exist.

V. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Mait Metelitsa,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis *A Functional Prototype and General Architecture of Analytic Data Management for a Railway Company*, supervised by *Kristo Raun* and *Prof. Ahmed Awad*.

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mait Metelitsa

15/05/2024