universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Cloud data storage framework for efficient and reliable content retrieval

verfasst von | submitted by

Arthur Rieß BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

Für Sabrina

Abstract

Cloud-based data architectures are a leading approach to handle the scale and variety of modern information requirements. This thesis explores the question how the data storage for the FactCheck framework, a system for automatic comparison of semi-structured data, can benefit from a migration to Azure's cloud infrastructure. The existing CouchDB implementation suffers from slow data retrieval due to limited indexing capabilities. Also, the usability for developers is negatively affected by split data access and unclear separation of concern. To update the existing architecture, requirements from stakeholders are collected and a cloud-native architecture is designed based on a literature review analyzing the state of the art in cloud data storage systems. A modular approach enables polyglot persistence. To test the design the cloud components are implemented and the necessary changes for communication to the FactCheck framework are performed. The resulting system facilitates faster queries while preserving all existing functionalities. A data transformation pipeline ensures data consistency and cleanliness while saving costs by using a self-hosted runtime, but performance remains insufficient for full production use. Still, the new implementation contributes to the success of the FactCheck system by improving performance and usability for users and developers.

Kurzfassung

Cloud-basierte Datenarchitekturen zählen zu den zentralen Maßnahmen, um der wachsenden Vielfalt und Menge moderner Informationsanforderungen gerecht zu werden. Diese Arbeit untersucht die Frage, wie das FactCheck-Framework, ein System zum automatisierten Vergleich semi-strukturierter Daten, von einer Migration in die Azure-Cloud-Infrastruktur profitieren kann. Die bisherige CouchDB Implementierung weist Einschränkungen bei der Abfragegeschwindigkeit aufgrund begrenzter Indizierungsmöglichkeiten auf. Zudem ist die Nutzbarkeit für Entwickler durch separate Datenzugriffe und eine unklare Trennung von Verantwortlichkeiten eingeschränkt. Zur Verbesserung wurden Anforderungen mit den relevanten Stakeholdern erhoben und auf Basis einer Literaturrecherche zu aktuellen Cloud-Datensystemen ein ein neues Cloud-Design entworfen und implementiert. Ein modularer Ansatz ermöglicht dabei polyglotte Persistenz. Die Azure-Komponenten wurden getestet und die Kommunikation mit dem FactCheck-Framework entsprechend angepasst. Das resultierende System ermöglicht schnellere Abfragen und unterstützt weiterhin sämtliche bestehenden Funktionalitäten. Durch die Nutzung einer selbstgehosteten Integration Runtime konnten Kosten bei der Daten-Transformation reduziert werden während Konsistenz und Datenqualität sicher gestellt sind, auch wenn die Performance für Einsätze in einem Produktions-Umfeld noch nicht ausreichend ist. Dennoch leistet die neue Architektur einen wichtigen Beitrag für den Erfolg des FactCheck-Systems, indem die Performanz und Nutzbarkeit für Nutzer und Entwickler erhöht werden.

Contents

List of Tables

List of Figures

Listings

1. Introduction

The advancement of technology has introduced a need for multiple revolutions regarding the processes with which data is generated, extracted, transformed and stored. The amount of data scientific projects and organizations face is unprecedented and expected to grow even further. Additionally, data is drawn from a variety of sources with specific formats and requirements for interaction and transformation. The underlying hardware that enables data sinks has seen its capabilities grow rapidly. Furthermore, the cloud allows for computing at scale through resource pooling and offers services that lift the burden of resource management of the end-user.
All these trends have caused approaches to model data to go through multiple evolutions with branching strategies for broad application or niche requirements. Traditional data achitectures struggle with different types of data such as unstructured or semi-structured formats from images, texts or audio. Another factor affecting legacy data models is the varying speeds at which data might arrive in a modern data architecture, such as real-time readings from sensors or random uploads in large bulks. As data becomes more relied upon in modern society for driving decisions, the analytical capabilities raw data needs to be prepared for diverges significantly. Enabling a variety of data analytic tools and use cases, such as Machine Learning (ML) and Artificial Intelligence (AI), is one of the main requirements of a modern data modeling approach.
In a modern data ecosystem, problems might start to creep up once a data model has been deployed over a long period of time. The data model might need to adapt to additional demands either generated by outside influences or from the project or business itself. Over multiple iterations of adaptation the data model might fracture and struggle to fully fill its intended purpose or meet its key performance indicators such as query performance or cost. Even models left unchanged can become less usable over time if its metadata is not properly maintained. For example, without sufficient metadata management users might not be able to locate the information they need.
Another noteworthy industry trend is fueled by a bottleneck in skilled labor capable of implementing and managing complex data models and leads to some solutions prioritizing simplicity, standardization and reusability.
As discussed in [50], more recent developments in data management are not yet fully understood by scientific literature due to a lack of experience with long-term, real-life implementations. This can lead to imprecision when discussing a modeling approach benefit and fitting use cases.

1.1. Motivation

This work was created in the context of FactCheck. FactCheck is an analysis platform for comparing facts available as semi-structured data. The Facts are scraped from the web and stored around entities. They can then be compared using precision metrics and served to end-users for example via a browser plugin called Idefix. Besides the backend data store which contains raw, cleaned and curated data, FactCheck consists of the middle layer encapsulating the core business logic and a frontend with a graphical user interface and an API.
With cloud computing and the requirements brought by big data, such as increased storage and complex analysis, the requirements towards data store models has expanded from the traditional data store to include advanced metadata management, analytical capabilities, efficient processing and more. The goal of the architecture is to facilitate cost effective processing and transformation as well as fast querying and analytics. Although most of the data will be initially ingested in semi-structured formats, to enable analytical capabilities structured data will be derived from transformation. Therefore allowing for polyglot persistence, the integration of multiple different data stores, will be crucial.
This motivates the thesis outlined in the following section.

1.2. Research Questions

The question of this master thesis is how to design and implement a reliable and efficient data storage framework on Azure to support content delivery and semi-structured data retrieval. The framework requires fast and customized database solutions for efficient data storage and access to support real-time look-ups and the generation of statistics. This project aims to compare different database solutions, including cloud-based solutions (e.g. Azure `CosmosDB`), and evaluate their suitability as reliable and efficient data storage for ongoing FactCheck projects. The evaluation will also include the current Apache `CouchDB` (NoSQL) database. The project will develop strategies for the most efficient access to data with respect to FactCheck's main use cases. This can be subdivided into these research questions:

1. How can a reliable and cost-efficient data storage framework for FactCheck be designed to support both content delivery and the retrieval of semi-structured data?

2. How can a Proof of Concept of this design for the storage framework be implemented?

3. Which constraints exist for such an implementation? Examples of such constraints may include - but are not limited to - costs and complexity.

4. How can a Content Delivery Network (CDN) support the FactCheck environment?

1.3. Outline

The basis for this work is a review of existing research on cloud data storage, CDNs and the functionalities provided by Azure services. As a first step in the approach for the data storage framework, the specific requirements for the framework have to be gathered and general requirements have to be refined and prioritized. For this purpose collaboration with stakeholders is key. As preparation, the current scheme of the database has to be analyzed to understand the necessary adjustments. It is expected that the stakeholders don't anticipate all the requirements and changes themselves. The advantages and disadvantages of the current `CouchDB` Implementation are to be discussed. Furthermore a focus is put on defining the term "efficiency" for the context of this work, for example between metrics such as memory used, costs incurred and scalability. Based on the most important requirements a baseline framework will be designed focusing on scalability, reliability and cost efficiency. Furthermore, the framework is designed with the requirements of supporting the existing project in the FactCheck environment. The next step is to develop and configure the data storage framework on Azure. The implementation will be managed and documented in Git. After the initial implementation the Factserver is adapted for communication with the data storage framework. To enable reproduction and Continuous Integration best practices the infrastructure will be version controlled as well. Part of the development is a high-level architecture diagram displaying core infrastructure components as well as data flow diagrams.
This implementation will be optimized iteratively, based on the requirements as well as stakeholder feedback. One method employed to verify the implementation of the requirements are performance tests evaluating the framework's scalability, and cost efficiency. The specific amount of iterations depends largely on the availability of stakeholders for feedback, therefore a minimum of 2 iterations is planned. The framework may include processes for methodologies which came up in user stories along the development such as data pipelines. The final step of the work must be a knowledge transfer session to enable a smooth handover to future end-users.
Out of scope for this work is an analysis of other cloud service providers besides Microsoft Azure. Additionally integration of existing application-level logic for content delivery will not be covered since the focus is on the data storage framework.
Based on this outline the following chapters are structured as follows: Chapter 2 introduces the theoretical foundations and the related work for the topics covered in the thesis such as differently types of data storage and content delivery in general. Chapter 3 covers requirements as well as an analysis of the current `CouchDB` implementation with its strengths and weaknesses. Chapter 4 covers the design of the cloud data storage framework and its components in Azure. Chapter 5 covers the implementation of the Azure components as well as the changes necessary for the Factserver. In Chapter 6, the implementation is evaluated based on available metrics and feedback. Chapter 7 provides thoughts on possible future work to improve the implementation that are out of scope for this work. Finally, Chapter 8 summarizes the thesis and provides a conclusion.

2. Theoretical Foundations and Related Work

This chapter lays the foundation for a cloud-native storage framework for large amounts of semi-structured and multimedia data scraped from the web. It was originally written as part of the master seminar and then adjusted for this thesis. The goal of the architecture is to facilitate cost effective processing and transformation as well as fast querying and analytics. Although most of the data will be initially ingested in semi-structured formats, to enable analytical capabilities structured data will be derived through transformation. Therefore, allowing for polyglot persistence, the integration of multiple different data stores, will be crucial.
This chapter is organized as follows. Section 2.1 presents the methodology of the undertaken literature review. Section 2.2 presents foundational theories on some of the most prominent methods in data storage and data management. Furthermore, related work to the most used data management approaches in literature is presented in section 2.3. In section 2.4 CDNs are highlighted as an important consideration in today's distributed data frameworks. Section 2.5 summarizes the information presented.

2.1. Methodology

This is a systematic review following the process outlined in [45]. This review aims to capture the current state of the art in data management, which provides an overview of the existing technologies and their strengths. To be eligible for inclusion in this review paper must be published after 2015 and be written in English. Papers were excluded if the domain they focus on is too narrow to draw conclusions towards the general state of the art of data modeling frameworks. Searches were limited to the subject Area of "Computer Science" and ordered by "Citated by (highest)". The information sources used are the library of the University of Vienna (https://usearch.univie.ac.at/), ieeexplore.ieee.org, Association for Computing Machinery Digital Library (https://dl.acm.org/) and scopus.com. The databases were last searched on the 30th of November 2024.
Three known relevant terms in data management were used as a starting point for this literature review. Based on these first results, further search terms are identified by looking at key terms in the paper, as well as referenced literature and citations. The search strategy consists of 10 iterations of search using the following filters:

1. Inmon OR Kimball OR "Data vault"
2. Gartner OR “Data fabric” OR “Data mesh” OR Lambda OR Kappa OR Lakehouse AND Data

3. "Data architectures" OR "Data landscapes" OR "Data Hub-Spoke"
4. "Analytics Store" AND Data
5. Kimball AND Data
6. Data AND Modelling
7. "Delta lake"
8. "Data lake"
9. "Data warehouse"
10. "Content Delivery Networks"

No tools were used to assess the risk of bias in the included papers. The papers are grouped around the most often mentioned data management methods such as data mesh, Data vault, Lambda and lakehouse and then summarized to capture their key points. 39 papers are included in the review. A lot of papers focus on one method. These can be separated into two groups where one is the founding paper for a method or an evolution of the method or secondly where the paper is an implementation process for a specific domain were the method is already chosen as a prerequisite for the paper. Other papers included also aim at providing at general oversight at the state of the art in varying aspects of the modern data landscape.

2.2. Data Store Models

In this section the results of the literature review are presented. In total 39 papers from 4 libraries were identified over 10 search-iterations. First, this section looks at those data store models which are well understood by scientific literature because they were introduced before the rise of cloud computing and plenty of work on theory and implementation has been done over the years. These are most likely to be relevant for the design and implementation of a cloud data storage enabling analytical capabilities from semi-structured data. Namely these stores are: Relational Database Systems, Key-Value Stores, Document Databases and Graph Databases. With cloud computing and the requirements brought by big data, such as increased storage and complex analysis, the requirements towards data store models eventually expanded from the traditional data store to include advanced metadata management, analytical capabilities, efficient processing and more. The literature review identified an increased interest in the concept of the Data Lakehouse (LH) which is a combination of the Data Lake (DL) and the Data Warehouse (DW). For this reason, these topics are covered after the foundational models.

2.2.1. Relational Database Systems

A Relational Database Management System (RDBMS) organizes data into structured tables defined by a schema that specifies column data types and constraints. The idea for data representation detached from the technical solution on the machine was first introduced in [9] in 1970, however relational databases still play a key role in modern data architectures. A primary key uniquely identifies a row in a table, while a foreign key establishes a relationship between two tables. Together, these keys ensure data integrity and reduce redundancy. RDBMSs allow users to query data via SQL. SQL is a declarative language, allowing users to express what they want from the database without specifying how to perform the computation. This aspect of s makes it suitable for the implementation of normalization, a process reinforcing data integrity by ensuring that information is stored only in one place (and thus reducing redundancy). On the other hand, strict adherence to normalization can lead to performance problems and, even more critically, a lack of evolution cast into the schema.
As described in [7], a common approach to integrate changes into a relational database system is using additional tables that transition new information onto existing objects. RDBMSs are an extremely generalized model that do not provide any implicit concepts enabling maintenance of an objects history. Depending on the relational model information on temporal aspects of transactions can be captured. The non-redundant attribute of normalized relational database systems clashes with the concept of completeness. Unless multiple versions of the data are stored it is not given that a query executed at any given time will produce the same result.
Entity-Relationship (ER) models play a foundational role in the design and implementation of relational database systems. They provide a conceptual framework for defining and structuring data before it is translated into relational schemas. ER models represent data as entities , relationships and attributes which makes them a useful tool for capturing the logical structure of data. This abstraction helps finding requirements, redundancies and keeping the model consist during database design. When converting an ER model to a relational database entities will be mapped to tables, attributes to columns and relationships to foreign keys. By offering a clear visualization of data, ER models allow for communication between technical and non-technical stakeholders.
While ER modeling focuses on the structural and logical representation of data, object-oriented (OO) modeling offers an approach that includes both data and behavior. In OO modeling, the concept of entities is expanded to include objects, which encapsulate attributes and methods. This aligns with modern programming paradigms and is often used for designing systems where behavior of the objects is as critical as the data structure. ER and OO have common strengths, such as the representation of entities and attributes, which can be used to connect database design and software development. For instance, entities in an ER model may correspond to classes in an OO model and relationships in the database can be represented as associations between objects.
A common method to design models for relational database systems is the Kimball modeling method described in [23]. It helps find a complete design that combines separate business processes. It relies on a rigid interview foundation to find the design questions

which drive the need of the enterprise. The nine-step design methodology is as follows:

1. Choose the process that answers the most important business questions and that is the most accessible for data extraction
2. Decide what a fact table record represents
3. Identify and conform the dimensions with the enterprise DW in mind. The long term strategy has to be considered here.
4. Choose the facts
5. Store pre-calculations in the fact table
6. Round out the dimension tables by adding as many text-like descriptors as necessary.
7. Choose how far back in time the fact table goes
8. Determine the need to distinguish between multiple snapshots of objects over time
9. Decide the physical design such as sort order of the fact table on the disk, the presence of pre-stored summaries or aggregations, backup and security.

2.2.2. Key-Value Store

A key-value store is one of the simplest data storage models. Data is handled as a collection of key-value pairs. Each key acts as a unique identifier, while the value can store any type of data, ranging from simple strings to complex objects like JSON or binary files. This model excels in scenarios where data retrieval is straightforward and involves direct access via a unique key, such as caching, session management, or configuration storage. Key-value stores are optimized for scalability and performance, such as in distributed systems and applications requiring low-latency operations with well-defined and predictable access patterns. Implementations offer variations in features such as persistence, in-memory storage, or scaling. However, key-value stores lack the flexibility of relational databases and are less suitable for complex relationships or structured querying such as joins.

2.2.3. Document Databases

Document databases are a type of NoSQL database designed to manage semi-structured data in the form of documents. NoSQL databases handle data which is not modeled in the table structure described in the previous chapters. Each document is typically stored in formats such as JSON or XML, and contains both the data and its schema. This allows for flexible and hierarchical data representations. This model is well-suited for applications requiring dynamic storage without schema-enforcement, such as content management systems, e-commerce platforms and real-time analytics. Document databases provide rich query capabilities, often enabling indexing and searching within document fields, which

makes them more versatile than key-value stores for complex data retrieval. Implementations include features like horizontal scalability, atomicity, consistency, isolation, durability (ACID) transactions and integration with distributed architectures. However, while they excel in flexibility and performance, document databases can introduce problems with schema evolution and keeping consistency across distributed systems, especially for highly connected or relational data.

2.2.4. Graph Databases

Graph databases are data store models designed to represent data as nodes, edges and properties, effectively capturing relationships between entities. This model is well-suited for applications where understanding complex relationships is essential, such as social networks, recommendation systems, fraud detection and knowledge graphs. Unlike relational databases, graph databases use graph structures to store data for efficient traversal of relationships without the need for costly joins. Graph databases offer features like built-in graph query languages and support for property graphs. They provide excellent performance for relationship-based operations, providing intuitive and efficient exploration of connected data. Graph databases can be less effective for use cases involving aggregation or unrelated data points, where other models like relational or columnar databases may perform better.

2.2.5. Data Warehouse

The idea of the DW is most closely associated with William Inmon. In [22] he defined a DW as a "subject-oriented, nonvolatile, integrated, time-variant collection of data in support of management decisions (see also: [21, 40])." According to Inmon, DW are distinguished by four key features:

1. Subject-Oriented: data is organized around key business topics, such as customers or products.
2. Integrated: data from various sources is transformed, standardized and stored in a unified format. This reduces inconsistencies.
3. Nonvolatile: data is stored long-term and remains unchanged, allowing historical comparisons and ensuring reproducibility.
4. Time-Variant: data captures changes over time, allowing analysis through timestamped records.

Insertion and analytics on the DW are enabled by data processing architectures such as the Lambda architecture, originally named "batch/realtime architecture" in [30]. It is a design pattern that combines online and batch processing to manage large-scale data applications with delayed data collection. It is structured into three main layers:

1. Batch Processing Layer: Pre-computes results on large datasets, enabling in-depth analysis of historical data.

2. Speed Layer: Real-time computations as data arrives, supporting immediate processing and response.

3. Serving Layer: Handles query requests by interfacing with the batch and speed layers, providing users with quick access to both real-time and batch-processed results.

This architecture allows efficient cost management by distinguishing which data processes require immediate computation versus batch processing. The Lambda architecture is especially useful for applications where streaming data must be validated before further processing as shown in [24]. Verified data is stored in a database, where batch scripts can later analyze patterns over time. Despite the challenge of skill demands for maintaining separate projects across data layers shown in [12], Lambda architecture effectively addresses big data challenges by supporting complex, multi-dimensional analysis while remaining cost-efficient.
With rising adoption and further development, data warehousing is set to remain essential for decision-support for data-driven industries.

2.2.6. Data Lake

Traditional data processing methods are often "schema-on-write" (for example extract-transform-load (ETL)). As described in [39], imposing schemas upfront can prove too rigid for increasing volumes of unstructured and semi-structured data. This motivated the development of DLs, which offer repositories for storing raw data in its original form. The term was first used in [56]. Unlike DW, DL use a schema-on-read approach. This preserves data flexibility and enables more versatile analysis. The cost associated with data transformation during ingestion is reduced, however the format of the data will have to be validated by processes on-read. The architecture of DL supports a flat structure where each data entity has a unique identifiers and metadata, which is important for targeted and on-demand querying for information as laid out in [49].
Two main DL architectures exist: zone and pond. In both architectures data is pre-processed.
In zone architectures, as described in [13], data moves through zones based on its level of refinement, starting in a raw zone and later progressing to standardized, cleansed, or pre-processed formats suited for particular use cases.
In pond architectures, data is stored in designated "ponds" depending on its characteristics. In this structure, original data is often lost as it is transformed, which diverges from the traditional DL philosophy of retaining raw data.
Some challenges modern DL face include high effort for data governance to prevent "data swamps", where poorly managed data loses value and metadata management for effective data discovery as shown in [42]. DL typically decouple storage from compute resources, which helps reduce costs associated with big data. However, this introduces challenges in ensuring data reliability and query performance.
DLs support a variety of enterprise use cases such as data science, Machine Learning (ML) and real-time analytics, integrating both SQL and NoSQL capabilities [12] [40].

Nevertheless, the effective use of DLs depends heavily on robust governance, structured metadata management and architecture that accommodates various analytical needs without compromising the quality or accessibility of the raw data.
[4] summarizes key lessons learned from building and operating an enterprise data lake. It is shown that scaling big data requires automating workflows such as data ingestion and the data recovery processes. This minimizes manual intervention and ensure platform reliability. While the DL introduces new tools, it must also support existing tools for easier user adoption and transitions. Furthermore, distinguishing between batch and user-facing processes allowed for optimized resource allocation and improved response times. [4] finds that traditional audits were insufficient for the DL environment. Instead, automated compliance mechanisms were necessary to manage data effectively. Finally, a single platform uniting all the access to data assets increased adoption from users and fostered new use cases. This shows the importance of centralized, accessible data repositories. The platform's design evolved based on user feedback and practical needs.
In [47] the design, implementation and operation of a corporate DL that has been running for over two years is described. It provides analytical capabilities for multiple teams within the company. The platform integrates different data assets, such as sales, marketing and customer information, from unconventional sources like weather and social media among others. One of the main challenges was managing data ingestion from geographically distributed silos. Another issue was enabling teams with varying technical expertise. To address this the concept of a "data asset" was introduced to streamline the data transfer, data ingestion and governance. The system uses Hortonworks Data Platform for storage and Kubernetes for managing control, utilizing components such as HDFS, Spark, Yarn and Zookeeper. The platform supports both batch and streaming ingestion, with batch ingestion offering more flexibility while streaming ingestion is continuous but more challenging to implement. Using this modular and flexible database integration, the DL successfully handles diverse data sources, although managing heterogeneous data ingestion stays complex.

2.2.7. Data Lakehouse

The model has emerged as an approach that combines the functionalities of DW and DL to meet modern data management needs.

Architecture

The LH integrates the data organization and governance strengths of DW with the scalable, low-cost storage of DL. Key features of the LH described in [19] include support for structured and unstructured data, metadata management and compatibility with big data analytics, Artificial Intelligence (AI) and ML applications.
The basic architecture is visualized in figure 2.1.

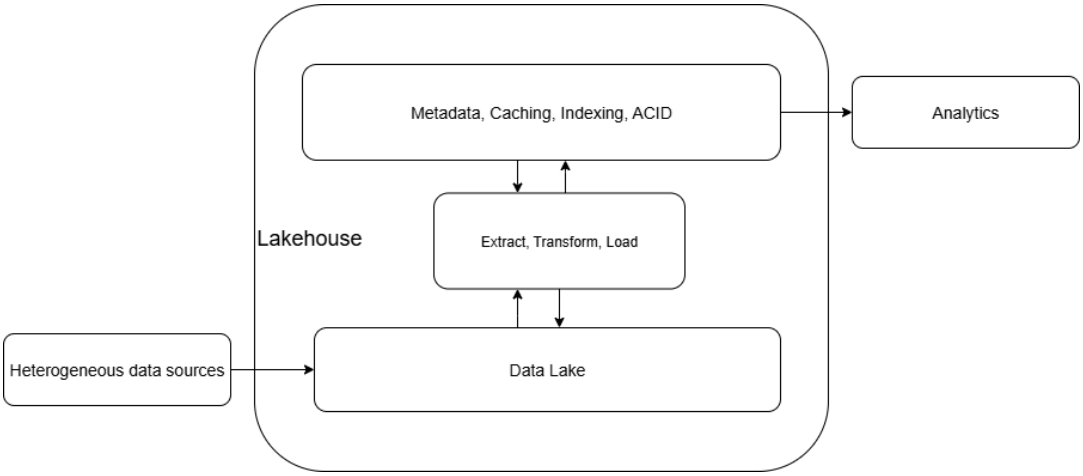

Figure 2.1.: Basic lakehouse architecture

Case Studies

[17] discusses how the evolution from DW to DL and then to LH, and addresses the limitations in earlier models. DW struggle with issues like high costs and the inability to handle unstructured data types such as images and the sensor data used in [55]. The LH mitigates these issues by providing a single platform that reduces ETL complexities, integrates a variety of data formats and enhances data accessibility through interoperability. Additionally, the LH architecture includes components for SQL analytics, addressing performance and query optimization while enabling ACID-compliant transactions.
Architecturally, the LH consists of data staging and historical storage areas similar to DL and analytical processing layers similar to DW. The staging area temporarily holds incoming data for processing, while the historical area preserves historical data with all changes, providing history on all data transformations. In the DW segment, layers such as the base and performance and analytics layers allow for refined data transformations and fast analysis (see also [50, 2, 44]).
In [43] for the purpose of simulating a unified interface to the end user a virtualization layer is deployed . There are further architecture ideas out there containing other combinations of zones atop of a data lake, for example zones where the raw data is "harmonized" using the data vault modeling approach, while another zone holds the raw data which might be modeled using a different method [43].
In [19], the state of the art technologies for DW, DL and LH are compared regarding their strengths and weaknesses. The paper focuses on their architectures, metadata storages for data organization, security aspects and data processing capabilities. The study includes a comparison between various LH implementations and service providers. LH systems are evaluated to identify the optimal fit for specific use case scenarios. A baseline conceptual architecture model for LH systems is proposed, aimed at improving metadata extraction during data ingestion, enhancing metadata management and storage

and increasing performance for online analytical processing (OLAP) queries. The proposed architecture integrates key features of DWs and DLs to create a unified, efficient and secure data management system. The study also implements and evaluates three SOTA systems: HDFS (DL), Hive (DW) and Delta (LH), chosen for their robust feature sets, community support and flexibility. These systems are compared through four analytical queries on the IMDb dataset, demonstrating their capabilities in handling large-scale data storage, processing and analysis. While Delta LH exhibits lower performance for simpler queries, it significantly outperforms HDFS and Hive in more complex query scenarios. Notably, its support for schema evolution, time travel (access to historical data) and efficient real-time analytics are mentioned as major strengths. Lastly, the paper identifies the shared limitations of DL and DW, particularly in handling high-velocity data. The study suggests tools for modern data requirements, such as improving the data ingestion processes and advanced metadata management.

In [3] the evolution and significance of DL architectures in modern data management and their benefits and challenges are discussed. DL excel in addressing the diversity of data sources, offering schema-on-read capabilities to manage and analyze vast volumes of unstructured and structured data efficiently. This technology enabled enterprises to move beyond traditional schema-first approaches, which lack the agility required for modern data integration. DLs simplify the integration of multiple data sources, reducing the reliance on proprietary solutions by using open-source tools and offering cost-effective and scalable solutions. They also enable real-time data ingestion for timely decision-making and analysis. However, early implementations of DLs were often built on tightly coupled storage and computing architectures such as Apache Hadoop and faced conflicts between deep analytics and low-latency applications. The maturation of DLs has positioned them as a cornerstone of modern data architecture, giving organizations room to adapt to new data sources and evolving needs of the business.

[3] also addresses the challenges of implementing and maintaining DLs, including data quality concerns, security and the lack of standardized schema or metadata governance. Without effective policies, DL risk becoming data swamps, characterized by poor data lineage and data quality. Metadata management systems and role-based security are important for mitigating these risks, especially for enterprises in heavily regulated industries. AI-driven automation is further transforming DL management by automating metadata generation and data cleaning, therefore making data analysis more consistent and accurate. Governance policies are needed to avoid losing oversight of vast data volumes. Robust protective measures are needed when multiple business units access the same DL. The tracking of metadata requires specialized software and expertise. To address these challenges integrating AI and ML for metadata extraction is proposed. Lastly, a consistent visual representation of modern DL architecture is presented. This enables enterprises to maintain scalability and efficiency in managing distributed data ecosystems.

In [51], an overview of concepts related to the LH is presented. The LH as an architectural approach is evaluated based on data analytics use cases, with a focus on technologies like Delta Lake, Apache Hudi and Apache Iceberg. The section "data warehouses and

data lakes" stresses the limitations of DW which lead to the emergence of DL. However, challenges in DL development and operation, such as architectural complexity, integration of batch and stream processing and issues of consistency, atomicity and isolation, demonstrate the need for novel architectural approaches like LHs. The key challenges in the DL addressed by the LH are:

1. Complex architecture: DLs often involve multiple storage systems for diverse data formats and a combination of batch and stream processing engines. This can lead to increased maintenance and error-proneness.

2. Combining Batch and Stream Processing: The traditional Lambda architecture requires separate batch and speed layers, resulting in duplicated application logic and increased maintenance efforts. Although the Kappa architecture reduces complexity by only using stream processing, it struggles with computationally intensive tasks. Modern engines like Apache Spark and Apache Flink partially unify batch and stream processing but face limitations such as inefficient handling of incremental data updates.

3. Incremental Processing: Immutability and periodic re-computation in the Lambda architecture create significant computational overhead. Incremental processing offers optimization potential but can not avoid all constraints of existing batch engines, such as lack of efficient mechanisms for partial updates.

4. Consistency Enforcement: Distributed file systems used in DLs lack built-in schema validation or enforcement, requiring manual implementation of consistency checks. This leads to additional development effort, boilerplate code and increased error risk.

5. Atomicity and Isolation: Ensuring ACID properties in DLs is challenging. Many processing engines, such as Apache Spark, do not guarantee atomicity. Additional measures, such as locking mechanisms or supplementary storage systems, are often required.

In [18] the desired features for the LH are identified by comparing the strengths of DLs and DWs. The properties found are:

- Elastic, highly available and cost-effective data storage
- Adaptability to varied and evolving data structures and their metadata
- Smart data transformation for processing and sorting
- Support for analytical queries with user-friendly tools

[51] mentions that LH are still in their early adoption and further research is needed. This discussion shows the need for new tools and methodologies to enhance data ingestion, transformation and access, ensuring that LH can effectively combine the strengths of DL

and DW while addressing their limitations.
In [8], the challenges with a legacy LH architecture at Meta like duplication, poor performance and inconsistent user experience are outlined. To address these issues, Meta started an organization-wide effort called Shared Foundations for the organization of its data infrastructure. The effort aimed to rebuild the data LH stack based on three core principles:

1. Fewer Systems: Reducing redundancy by combining overlapping systems into single solutions.

2. Shared Components: Instead of a one-size-fits-all system, different compute engines for distinct use cases such as batch or interactive querying were developed. These engines share reusable components to promote efficiency.

3. Consistent APIs: Ensuring a unified user experience by providing consistent APIs across different systems.

The Shared Foundations program brought several advantages. Because multiple systems were combined the engineers were able to focus on fewer projects, enabling faster development and allowing teams to concentrate on developing new features and optimizations. Users also benefited from the consistent system, by reducing the learning curve when switching between systems. Furthermore users benefited from a better system performance. The resulting architecture streamlined data integration and minimized components. Several technological developments created during this initiative were made open-source, enabling broader adoption and collaboration.
In [5] a LH is implemented in the context of biomedical research and health data analytics. The LH focuses on features for the unique requirements of this domain, such as advanced access control models. The LH architecture evolved from addressing project-specific data management needs. Data is sourced from relational and non-relational sources, transformed to open formats like Parquet and made available for relational querying and ML applications. The architecture ensures that data is up-to-date, minimizes ownership cost and supports lifecycle management. The adoption of Parquet-based data formats resulted in significant storage reduction and performance improvements. For instance, a dataset of 50.8 million rows saw a nearly ten-times reduction in file size compared to CSV. The same dataset also experienced query performance improvements during analysis. The LH reduced storage costs, enhanced query performance and was rapidly adopted by research teams. These benefits show its fit for managing massive, heterogeneous datasets in biomedical contexts. This case demonstrates how adapting LH features to domain-specific requirements can improve large-scale data management in specialized fields.
[54] presents the design and implementation of an open data platform based on the LH architecture for network telemetry data. The platform follows the cloud-native principles, enabling deployment in both managed and self-hosted environments. Interoperability is achieved by utilizing standardized data ingestion formats and the implementation is provided as open-source. Raw data is ingested in telemetry formats or JSON. Data is

then normalized and converted into structured Avro records using Spark streaming jobs and stored in Iceberg tables. Stored data can be accessed via the Trino SQL engine, further processed with Spark batch jobs, or visualized through Apache Superset's web UI. Furthermore, the platform supports batch and streaming jobs with Apache Spark and integration with Python for automation. This prototype demonstrates how a LH can handle the challenges of network telemetry data while maintaining flexibility and scalability using open-source.
[57] presents the Medical LH, a platform designed to integrate medical data from varying data sources into a unified data format. This supports medical analysis by reducing data acquisition costs. The architecture leverages cluster computing for processing multimodal data and consists of six layers:

1. Data Ingestion Layer: Migrates large-scale structured and streaming data to Hadoop Distributed File System (HDFS) and standardizing data for storage.
2. Data Storage Layer: Stores multi-modal data in HDFS, acting as a bridge between the DW and DL.
3. Data Management Layer: A DW Module for efficient querying and a DL Module to unify formats and handle structured and unstructured data.
4. Data Scheduling Layer: Either persistently stores data or processes data in memory without permanent storage.
5. Ecological Service Layer: Ensures cluster coordination and communication.
6. Application Layer: Provides Restful APIs for user interaction and visualization.

The Medical LH allows for querying of heterogeneous medical data, supporting medical decision-making while maintaining scalability and usability.

2.3. Data Management Approaches

This section explores data management approaches that have developed for advanced data stores in the cloud. Over the last years this development has been mainly driven by industrial stakeholders. The three approaches covered are data vault, data mesh and data fabric.
Data vault modeling is a method for structuring data in a way that aligns with the data-driven and flexible approach proposed by Inmon in [22]. [22] argues that requirements for a DW cannot always be known before the DW is modeled and that all available data should be stored and accessible to support business decision-making processes. Data vault supports this by loading raw data from source systems into a central data store without transformation, ensuring that data remains traceable and adaptable to change [59]. The open data vault method data vault was described in [27].
An example Data Vault Model is visualized in 2.2. The model is highly flexible, consisting

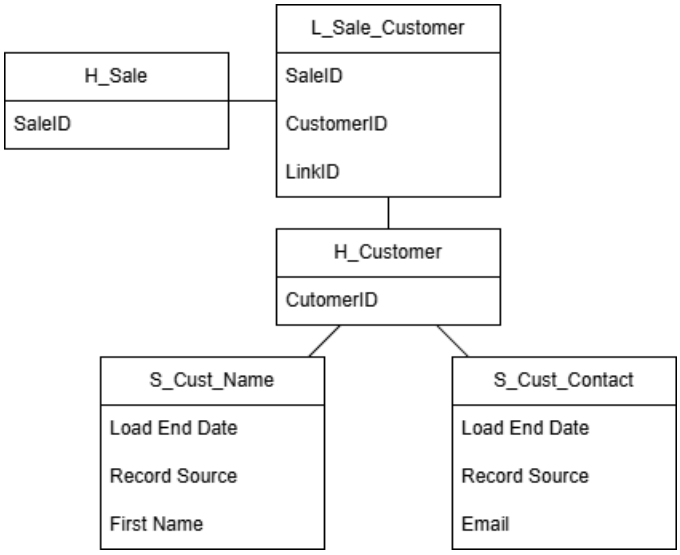

Figure 2.2.: Example Data Vault Model

of three core table types: hubs, links and satellites. Hubs store unique business keys, representing core business objects. Links define many-to-many relationships between hubs, allowing for complex connections. Business objects have a stable identifier and can be arbitrarily structured. This makes them suited for tracing changes in attributes since the structure of an object is decoupled from the object itself as well as the relationship itself being always created as a table. As discussed in [7] obtaining the history of the schema is not possible. Satellites provide descriptive details for each hub entry and are designed to capture changes over time. This allows historical tracking and adaptability. Data vault incorporates "Load Date" and "Load End Date" to naturally capture transaction time [7].
Unlike dimensional models, which are built to address predefined business needs, data vault is structured for evolving requirements. There is minimal transformation needed during loading, which, as discussed in [14], enhances both performance and traceability . To enable full traceability of all transactions the source of each transaction is recorded. If the structure of the source changes, a new Hub will be introduced referring to the original Hub through a "Same-as" relation [7].
Data vault modeling does not try to capture business processes but instead tries to capture the entire enterprise organization in it's model. One key advantage is the fast initial data loading, which is possible since no transformation is done at this stage.
First described in [10], data mesh is a decentralized, domain-driven approach to managing organizational data at scale, introduced as a solution to limitations of monolithic data architectures such as DL and DW. The four design principles are: "data-as-a-product, domain-orientation, self-service platforms and federated computational governance". This shows the core concepts of consumption-ready data products to be distributed for consumption by other business domains (see also [6]).

Data-as-a-Product treats data not just as raw information but as a product intended to meet the needs of internal and external consumers. Data product owners are responsible for quality assurance, data accessibility and Service-Level-Agreements helping with the prioritization with reliability in data.
Distributed data domains emphasize domain-specific data management, allowing teams with specialized knowledge of their business area to oversee their data independently. This localized control enables domain teams to tailor their data solutions more effectively while reducing the bottleneck created by central data teams.
[28] describes how self-serve data platforms enables domain teams to manage their data products, supported by a shared infrastructure that reduces redundancy and establishes interoperability. This platform provides tools for data ingestion, processing and serving. Redundant technical effort can be removed if domains can reuse products from other domains[46].
Federated Computational Governance implements a governance model that balances centralized oversight with domain autonomy. This principle enforces consistent policies across data products while allowing domain teams the flexibility needed for agile development (see also [6] [11]).
[15] describes how data mesh addresses critical organizational challenges, particularly the scalability constraints and bottlenecks associated with central data teams, as well as high costs of discovering data inside an organization and maintaining this overview. However, it also introduces challenges, including the need for skilled domain teams, potential data duplication and increased management costs. A prerequisite for successful installation of the data mesh model is a mature data platform with well defined data governance.

Data fabric is a modern approach designed to address the complexities of big data integration and management, specifically for enterprises needing fast analytics and company-wide access to all data assets. Central to data fabric is a knowledge graph acting as an enterprise knowledge base. It contains all business-relevant entities and their relationships. This graph maps these entities across various data sources, streamlining the integration and enrichment of new data by automatically aligning it with existing knowledge as described in [26]. Key capabilities of data fabric include:

- Machine Learning Integration: ML enhances data usability and decision-making from data collection to optimization.
- Unified data Integration: Data fabric links all sources (file storages, DBMSs, DLs) into one platform through APIs, ensuring a cross-system data flow.
- Modular architecture: Built with microservices to offer flexibility and scalability.
- Secure Multi-User Access: Customizable access permissions support role-based data access.

The data fabric concept focuses on metadata and polyglot persistence without promoting concrete implementation paradigms. It helps create reusable and flexible data integration.

The knowledge graph includes data lineage in the form of a description of collection and processing activities. Knowledge graphs can be created using ML technologies. As with data mesh reusability of data products and structures like Pipelines lead to a more agile operation (see also [6], [46]).

2.4. Content Delivery Network

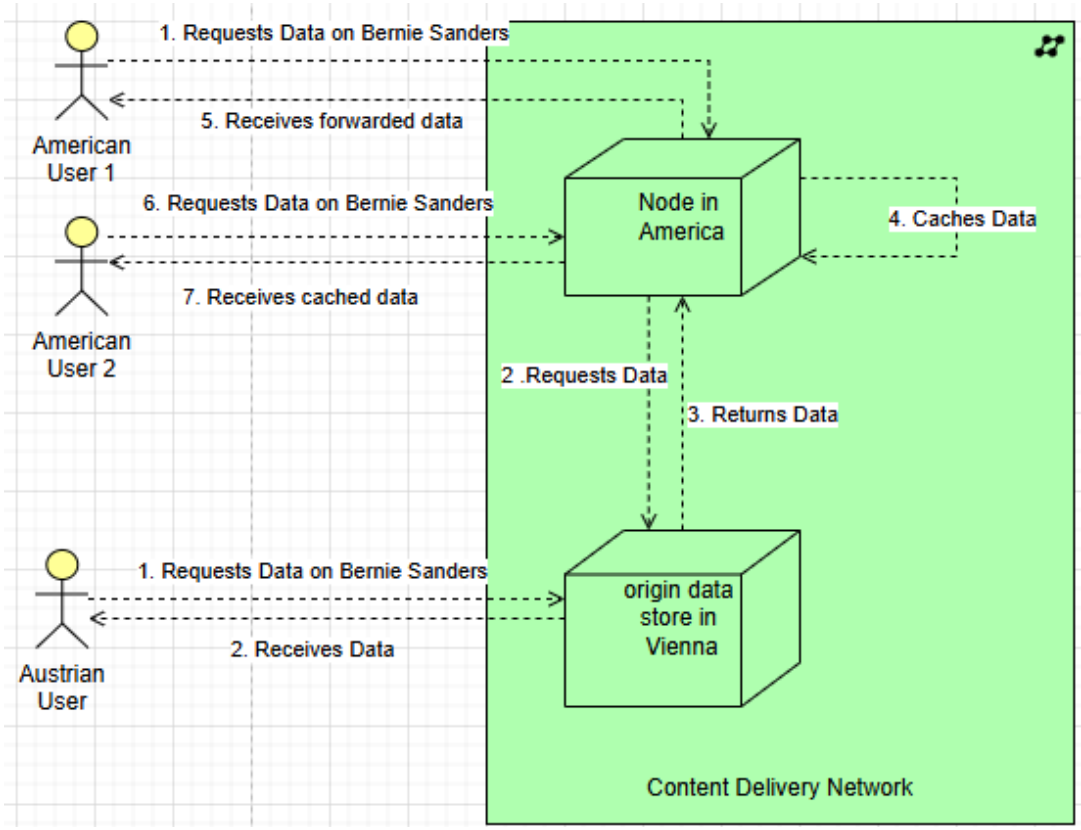

Figure 2.3.: Example CDN workflow

CDNs are distributed servers placed across different locations close to users to support the delivery of data. They play an important role in modern, global data architecture to improve performance and scalability. When a user connects with a webservice supported by a CDN they are routed to the closest available proxy server. This server either already has the requested data cached or will request it from its source and cache it in the process. The data will be stored until a preset time-to-life (TTL) expires. A visualization of an example workflow using a server to cache content can be seen in figure 2.3
[20] discusses pros and cons of deploying CDN. A CDN will reduce the average latency for users by reducing the delay in server response-time and in the delivery of packets.

One disadvantage is the reliance on third party infrastructure providers, since most if not all organizations do not have the means necessary to have global, duplicate nodes to replicate their data.
In [58] the state of the art on CDNs is discussed. Literature is collected and classified into areas of research. It is concluded that industry is leading the development of the technology and academia is following the trends leaving room for additional research in scientific literature.

2.5. Summary

This chapter explored different data stores and varying data management approaches and focused on literature targeting the design and application of cloud data architectures across various domains, highlighting their ability to integrate structured, semi-structured and unstructured data efficiently. The literature examined serves as the foundation for the proposed cloud native storage framework for storage and processing of vast amount of semi-structured data and polyglot persistence.
The findings show that there is a multitude of data management methods with different applications. The best method for each use case can be difficult to determine because of the complexity of the field as well as the overlap between strengths of the methods. The LH combines the scalability and low-cost storage of DL with the robust querying capabilities of DW, addressing modern big data challenges. Such as the ability to handle heterogeneous data and to handle performance sensitive domains via reduces query latency and storage costs. DL can be extended to cater to specific domains, such as access control models for medical data or AI analysis for maritime applications.
There is study risk of bias as the literature review only went through four databases, which could be too little to capture all the trends in research. This could lead to imprecision. Furthermore the literature is heavily skewed towards the topic of the LH, especially regarding recent development, leaving other topics underrepresented.
There was no funding for this review. This review was not registered.

3. Conceptual Foundation and Requirements

This chapter derives the requirements for the design of a cloud data storage framework in Azure based on an analysis of the shortcomings of the current CouchDB implementation. The concepts representing the foundation of this framework are covered.

3.1. Analysis of the Current Data Storage Implementation

This section covers the legacy data storage solution CouchDB.

3.1.1. CouchDB

Apache `CouchDB`[1] is an open-source NoSQL database that uses document-oriented storage as described in section 2.2.3. Unlike traditional relational databases, `CouchDB` stores data as self-contained JSON documents identified by a unique `_id`. It is designed to be schema-free which allows flexible data structures. This can for example facilitate schemas which evolve over time.
`CouchDB` provides availability and reliability through features such as multi-master replication and crash-resistant storage. It also exposes a HTTP API, making it easy to integrate with web applications and RESTful services. Queries can be executed using MapReduce views, but these must be explicitly defined.

Pros

- Documents are stored as JSON objects without a predefined schema which allows dynamic and heterogeneous data.
- Support for HTTP requests makes integration with web services and front-end applications easy.
- `CouchDB` uses an append-only B-tree storage that minimizes the risk for data corruption.
- Automated replication allows easy setup of distributed systems and backups.

[1] `https://couchdb.apache.org/`

Cons

- All queries must be predefined as MapReduce views.
- Dynamic fields are difficult to index efficiently because indexes must be explicitly created.
- View indexing can be resource-intensive on large datasets or when many views are defined.
- `CouchDB` lacks support for cloud features such as auto-scaling and managed security.

3.1.2. Implementation in the FactServer

The access to `CouchDB` is mainly provided through a Database class which implements a singleton pattern and established the connection to the `CouchDB` instance with values provided by the users in a configuration file. The class provides methods for saving and deleting objects, retrieving objects by `id` and execution of views by providing the name of the view as well as `CouchDB` specific parameters such as `limit`, `start_key` and `include_docs`.
One important problem in the current `CouchDB` implementation is that the connection between the Factserver and the database is divided into four classes that create their own instance of `CouchDB` servers as highlighted in figure 3.1. For example, there is a general Database class, but also `Stats` and `ExtendedStats` classes that access the same `CouchDB` instance but use different databases or views as shown in figure 3.2. This leads to duplication of database logic and unclear boundaries of responsibility. While the separation of concerns into classes is good practice, a cleaner solution would abstract away the database logic from the user. Furthermore a Mango class is used to directly execute Mango queries without using for example a shared interface or base class. This mix of different ways to access the data makes the code difficult to follow and maintain. Another issue is that in order to follow the logic of the code in-depth knowledge of the views and their behavior in `CouchDB` is necessary. This raises the entry barrier for new developers to work with the system and forces developers to write overly verbose comments in comparison to for example more self-explanatory SQL statements.
In addition, the format of the view results is not well suited to be further processed in Python. The data is returned in a nested format with fields like rows, key and value, which makes parsing more complex. One example of this complexity is shown in listing 3.1.

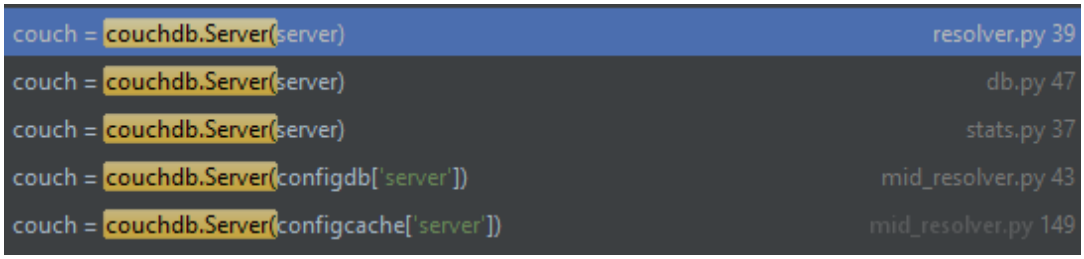

Figure 3.1.: Separate `CouchDB` instances

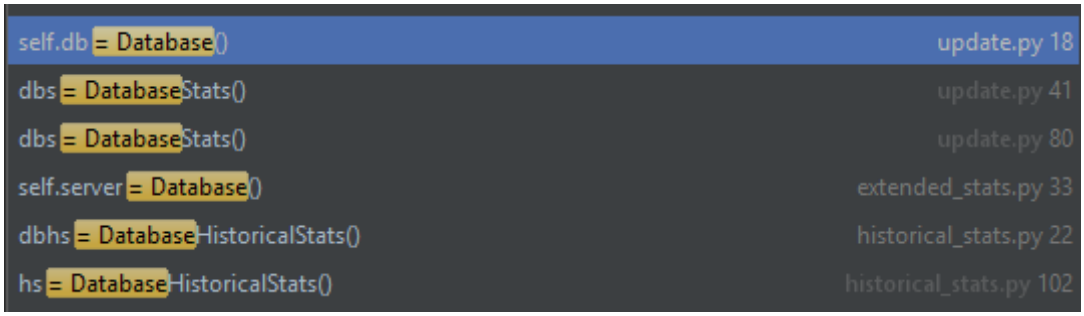

Figure 3.2.: Multiple Database Constructors

```
try:
    if len(result.rows) > 0 and aggregate:
        return result.rows[0]['value']
    if len(result.rows) > 0:
        return list(map(self.__map_result, result.rows))
except ResourceNotFound:
    if aggregate:
        return self.get_default_aggregated_stats()
return None
```

Listing 3.1: Example of parsing query results in `factserver.extended_stats.py`

3.1.3. CouchDB Factset

While `CouchDB` is capable of storing Factsets of any schema, most documents follow a similar structure to comply with the requirements of the different Factserver use cases. Those fields are shown in the following list, with indentation used to highlight membership of a field to the JSON mentioned above it :

_id Unique ID of the document.

mid Identifies the entity using the Google Knowledge Graph API[2]

[2] `https://developers.google.com/knowledge-graph/`

source The URL where the Factset was retrieved from.

data A JSON dictionary holding information on the entity.

name The name of the entity.

@type The type of the entity, for example 'Person'.

stats A JSON dictionary holding various statistical calculations for the Factset.

fact_stats A dictionary containing information on each column in the data object.

date Last time the document was updated.

However, the stats dictionary might not hold a value for some documents. The data and stats dictionaries might hold a varying additional amount of fields. Especially for the data dictionary the size can have large variety between documents.

3.2. Requirements

This section lists the requirements for the cloud data storage framework based on the analysis conducted in the previous section 3.1. The requirements are a combination of generic requirements that can be applied to most cloud storage frameworks and specific requirements targeting the needs of the FactCheck environment.

3.2.1. Functional Requirements

FR-1 The system shall perform efficient indexing to enable high speed data retrieval.

FR-2 Semi-structured data shall be stored in a cost-efficient manner at scale.

FR-3 To fulfill the previous two requirements multiple data sources shall seamlessly integrated.

FR-4 The system shall enable efficient and high-performance data retrieval mechanisms on global scale.

FR-5 The system shall provide interfaces for the existing content delivery and data retrieval mechanisms implemented in the Factserver.

3.2.2. Non-functional Requirements

NFR-1 The system shall handle an increasing volume of data without performance degradation.

NFR-2 High availability and fault tolerance shall be provided by the system.

NFR-3 The system shall be optimized for low-latency data access and high throughput with load peaks.

NFR-4 The cost of storage and compute resources shall be optimized.

NFR-5 Data retrieval and processing shall be able to fail gracefully on error and recover into a usable state while providing verbose logging on the incident.

4. Design

In this chapter the architecture of the new data storage framework will be presented.

4.1. Limitations

The design decisions presented in this thesis are subject to several limitations:

1. As this work was conducted within the scope of a Master thesis, the available time for implementation was limited. The result is that topics such as maintenance considerations, integration testing and performance optimization could not be fully covered.

2. The design is impacted by the cost limitations of the environment. Many Azure services, such as `Data Factory` and `CosmosDB`, can incur high operational costs. This limitation is one of the key drivers in the choice of technologies and the size of experimentation possible.

3. Tying into the previous point, the system was tested using reduced datasets and simulated scenarios, which may not reflect performance under actual production loads. Some assumptions made about scalability, cost efficiency, or general system behavior might not hold when deployed to a production environment.

4. While the system provides basic secure storage thanks to default Azure functionalities, advanced security mechanisms are out of scope due to their complexity. This topic needs to be covered for production deployment.

5. Cloud tools and their best practices evolve rapidly. The work presented reflects the state of Azure services at the time of writing. It is possible that more efficient alternatives become available soon, which might invalidate the decisions made for this implementation.

4.2. Proposed Design

This section introduces the new design of the storage framework.

4.2.1. Factserver Architecture

Figure 4.1 illustrates the overall architecture of the new Factserver system and its interaction with the Azure-based data processing pipeline. Factsets are uploaded into

the Raw Bucket of the Azure Data Lake, triggering a `BlobCreated` event that is picked up by Azure Event Grid[1] and forwarded to Data Factory. Data Factory orchestrates the separation of these files into structured components, which are stored as parquet files in the Enriched and Stats Buckets by the Factserver. Additionally, a domain summary file is written to support domain-level statistics. The Factserver also ingests the metadata into the Factsets container in `CosmosDB`. Using these components, the Factserver provides endpoints to manage, transform and analyze the data. This architecture supports modular processing and event-driven data ingestion while ensuring scalability and reliability.
Aligning this architecture with the LH researched in section 2.2.7, `CosmosDB` covers the DW functionality, Azure Data Lake functions as the DL and the LH engine that combines both components are implemented by the ADF and the Factserver.

4.2.2. Factset Schema

The structure of the Factset has been refactored to better align with the needs of the updated Azure architecture, while maintaining continuity with the previous `CouchDB` schema described in section 3.1.3. Core fields such as `MID`, `name`, `source`, `date` and `userid` remain unchanged. To remove CouchDB-specific naming conventions, the `_id` field has been renamed to `id`. `CosmosDBs` `_rid` field was excluded from Factserver logic to prevent similar platform-specific dependencies in the future. One addition is the new top-level `type` field, which holds the `@type` value from the data dictionary. Because this property is essential for downstream logic such as `MID` resolution and data partitioning, its promotion to a main attribute supports more efficient access and simplifies Factserver logic. Furthermore, the previously unified `stats` object has now been split into two separate dictionaries, named `stats` and `extended_stats`. While this results in some data redundancy and adds complexity to the Factserver logic, it enables a cleaner separation between different, project-specific calculations. In the former schema, these values were intertwined and methods targeting `extended_stats` could overwrite existing ones from `stats`, without transparency for the end-user.
Figure 4.2 visualizes the new schema and the storage location associated with each field. While the `fact_stats` object is part of the `extended_stats` object, since it is still a complex dictionary with a 1:1 relationship to the Factset it is more efficiently stored as its own Parquet file in the DL.

4.3. Design Components

This section introduces the components that enable the proposed data storage framework.

4.3.1. Data Ingestion and Cleaning

For the design of the data ingestion and cleaning workflow for this project, one architectural decision involves the deployment strategy for the data processing pipelines. The general

[1] https://learn.microsoft.com/en-us/azure/event-grid/overview

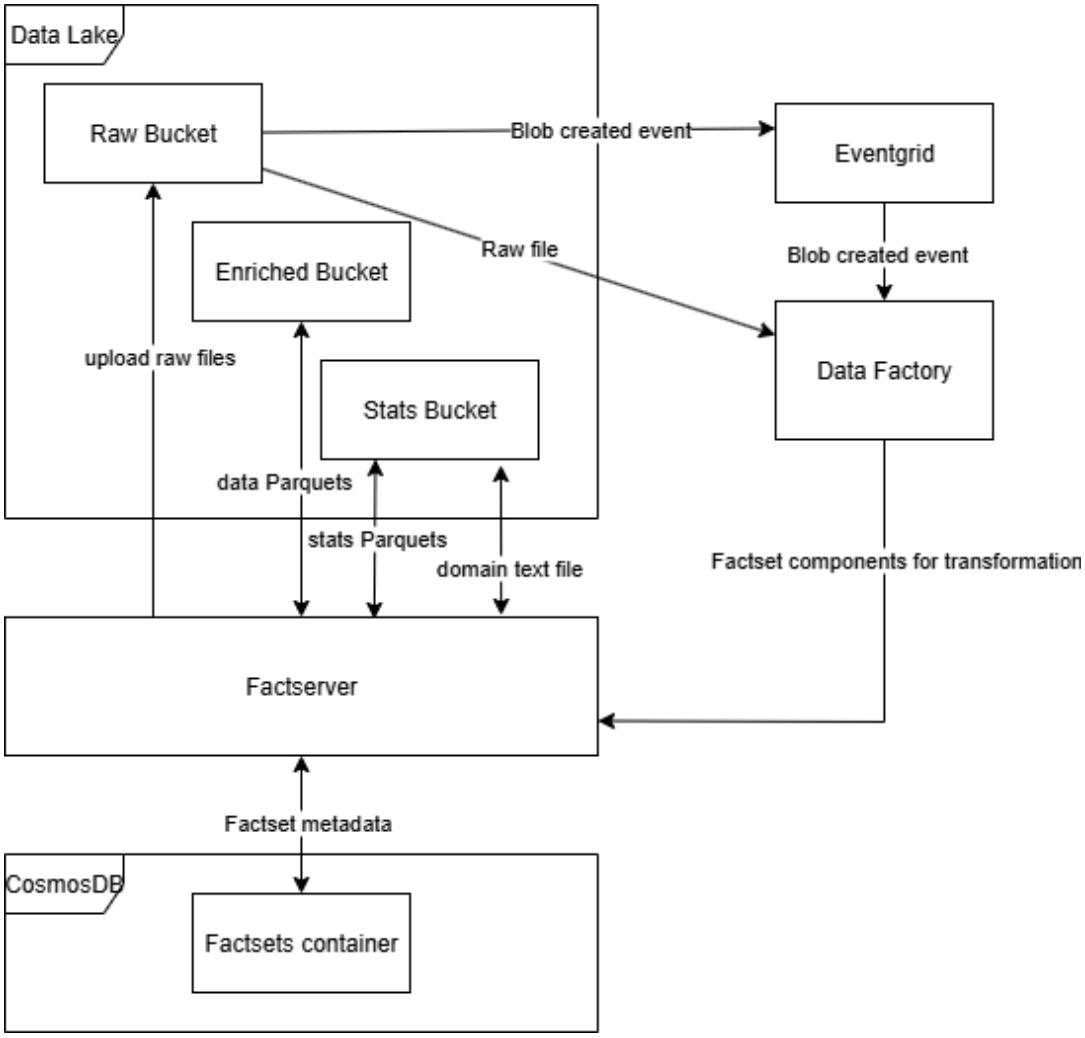

Figure 4.1.: System landscape for the storage framework interactions

approach involves first landing raw data into a cloud-based data lake, then performing cleaning operations and finally sinking the cleaned data into a data store (Azure `CosmosDB`). ADF was selected as the primary orchestration and processing tool for its integration with Azure services and flexible pipelines. The processing can be started by a trigger registered with a `BlobCreated` event from Azure Event Grid.
Given the cost requirements for the project, there are two options for optimization of the data transformations: 1. Use Azure's fully managed IR or 2. Deploy a SHIR hosted on university-owned infrastructure. Azure now allows hosting ADF IR on private servers ([36]), potentially reducing cloud-based compute costs.
The ADF billing model distinguishes between orchestration charges per pipeline, or activity run and execution charges per compute time spent moving or transforming

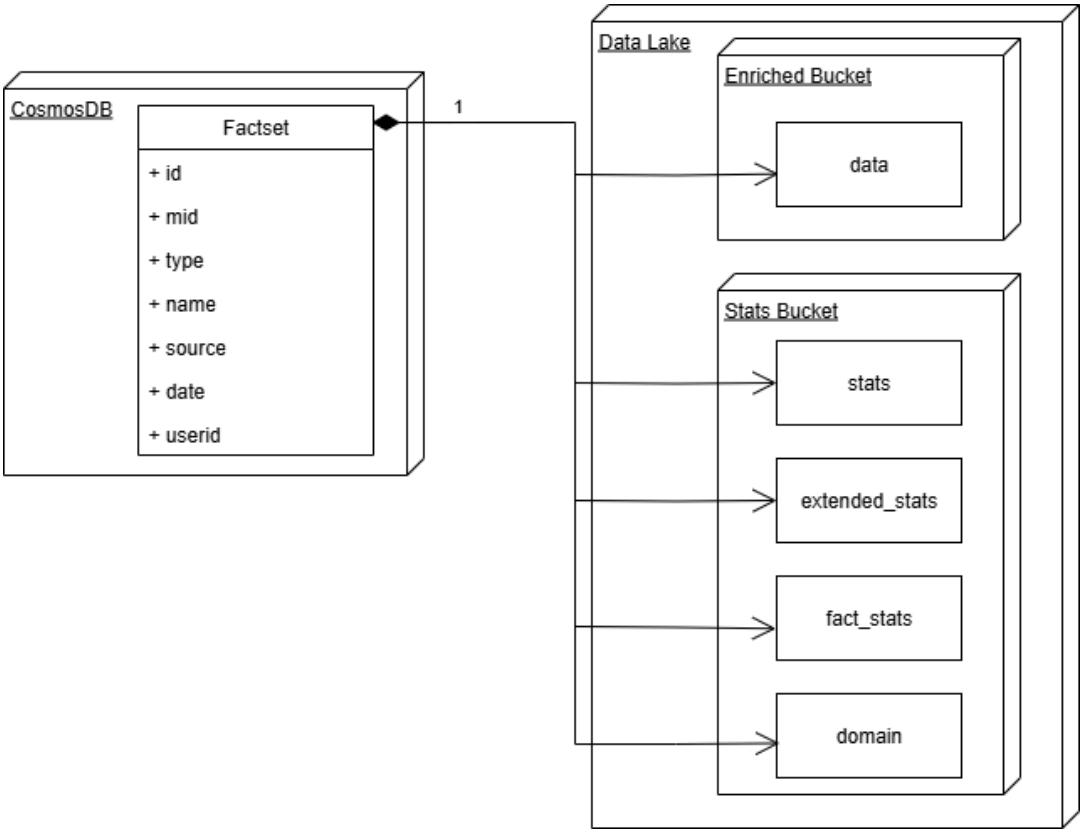

Figure 4.2.: Factset schema with associated storage location

data. While orchestration charges are low in both IR and SHIR, cost differences could be generated in compute costs. Data movement and transformation activities with IR cost around €0.232 per Data Integration Unit hour, while SHIR compute usage costs €0.093 per hour, a reduction of over 50%. Also, pipeline charges are lower using SHIR as described in [37].
In the context of this project, while data volume and run frequency might scale up in the future, it is essential to note that the compute demands for the foreseeable future can be fulfilled by university infrastructure already present. For this reason, no additional costs for this infrastructure are considered in this comparison. In section 6.1, the two setups will be tested to quantify the cost differences between them.

4.3.2. FactServer

In this subsection the changes to the design of the Factserver for communication with the data storage are discussed.

Azure Facade

Two design patterns were chosen for the interaction with Azure: the Facade[2] and the Singleton[3] pattern. While there was a single database class for the `CouchDB` implementation, many different projects implemented new classes and different methods to access the database as discussed in section 3.1.2, leading to a fragmented database layer with nested and complex dependencies. The Facade pattern is used to simplify and centralize access to Azure services such as `CosmosDB` and DL buckets. Using this pattern, application logic can interact with the storage layer through a single-entry point. While it is difficult to predict the complexity which might arise in the future once enough different developers have added their own methods to the Facade, refactoring and stream-lining processes from this dedicated entry should be easier.
As with the `CouchDB` implementation, the Singleton pattern ensures that only one instance of `AzureFacade` is used throughout the system. Since database connections and blob clients are expensive to create and hold internal state such as credentials and caches, sharing a single instance avoids unnecessary overhead.

Data Flow

Figure 4.3 illustrates the data flow through the internal modules of the Factserver. The diagram highlights the separation between the application logic and the underlying data infrastructure. At the bottom, the Database layer acts as the unified interface to persistent storage. This helps the core Factserver logic to gain independence of the storage backend. Communications through the `AzureFacade` and `Factset` class are grouped and highlighted as blue and green respectively. Above this layer modules such as `FactManager`, `Resolver` and `StatsManager` handle data transformation, enrichment and domain logic. This layered design supports maintainability and ensures that changes to the storage layer have minimal implications for the rest of the applications.

4.3.3. CosmosDB

Azure `CosmosDB`[4] is a globally distributed, Not-Only SQL database that offers high availability, scalability and low-latency access. In this project, `CosmosDB` is used as the primary storage for structured metadata, such as Factset identifiers, `MID`s, names and sources. `CosmosDB` organizes data into containers logically and partitions physically. Documents are stored in JSON format and queried using a SQL-like syntax.
Partitioning is used to ensure scalability and fast lookups. Based on the Factserver access patterns and after discussion with the stakeholders, `MID` was chosen as partitioning field. `CosmosDB` supports multiple consistency levels. The architecture uses the default session consistency[5], which guarantees that a client always sees its own writes and is cheaper and

[2] `https://refactoring.guru/design-patterns/facade`
[3] `https://refactoring.guru/design-patterns/singleton`
[4] `https://learn.microsoft.com/en-us/azure/cosmos-db/`
[5] `https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels#session-consistency`

faster than strong consistency[6], which would require replication to all regions before writes are confirmed. Indices are generated automatically, and per default one is calculated for every field in the container. To save the costs and time for computing indices which are not needed by the Factserver, currently the container is configured to just compute indices for `MID`, `type` and `name`. `date` could be a valuable index in the future but is currently rarely used in the Factserver. `userid` is not used in the Factserver for queries. `source` is sometimes used to query documents, but since field contains the entire URL and not just the domain there is high cardinality in the field making it a bad fit for indexing. The entire configuration is shown in listing 4.1.

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/mid/?"
        },
        {
            "path": "/name/?"
        },
        {
            "path": "/type/?"
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        },
        {
            "path": "/\"_etag\"/?"
        }
    ],
    "fullTextIndexes": []
}
```

Listing 4.1: CosmosDB indexing configuration

In `CouchDB` some projects experimented with the separation of concern in the schema by storing information in different databases in the same `CouchDB` instance. The separation of statistics in a dedicated database was already abandoned. One 'Cache' database was used as a slim redundant storage for `MIDs` to speed up the retrieval from the Factserver. With the introduction of the DL to outsource storage of large objects from the database, this use case ceases to exist. Therefore, a single unified container is used in `CosmosDB`. Internally, each document includes system-generated metadata such as `_ts`, `_etag` and `_self`, which can be used for versioning and change tracking. These are currently unused

[6] https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels#strong-consistency

in the Factserver, but `_ts`, which is the system timestamp, could be interesting for future developments for example for clean-up tasks in development.

4.3.4. Data Lake

This section covers the design of the DL technology and implementation.

Azure Data Lake Gen2

Azure Data Lake Storage Gen2 is a cloud storage solution designed for big data analytics workloads. It combines the scalability and performance capabilities of Azure Blob Storage with hierarchical namespace features, such as straightforward directory-based operations like renaming and deleting folders, known from traditional file systems. While a blob in Azure Blob Storage might have a name such as `/folder1/folder2/file`, these folders are just simulated for human users and are not actually represented on a technical level. On rename, for example, Azure Blob Storage is unable to just rename a single folder but instead must iterate over every file to check if it contains the folder name requested to be renamed and then rewrite the identified files. In the context of this architecture, hierarchical namespace is mainly used for partitioning.
Microsoft offers a preview package, `azure-storage-file-datalake`[7] that includes new directory level operations specific to hierarchical namespace, which there not possible in the mature `azure-storage-blob`[8] package. This package is chosen for the project to benefit from this development, for example for efficient listing of files in folders.
For the development setup, the Cool tier[9] was selected to balance lower storage costs with acceptable retrieval speed. It offers a cost-effective middle ground between the Hot tier, which is optimized for frequent access, and the Cold tier, which provides the lowest storage cost but with significantly increased access cost.

Partitioning

To organize the data efficiently and enable faster access, the files in the data lake are partitioned using a simple directory structure based on the entities `type` and `name`. Each Parquet file is stored under a `{type}/{name}.parquet` path. This approach provides an intuitive grouping of similar entities and allows for straightforward filtering based on the file path, which can significantly reduce the volume of data that needs to be scanned during queries. At the same time, the combination of information associated with the same `name` into one file reduces the number of smaller files and enables efficient compressing and storage. Another benefit is the reliability of the `type` and `name` field, for example the `MID` also conveys much information about the entity, however it might not be specified for certain `type-name` combinations or in other use cases the `data` might need to be accessed while the `MID` is not yet resolved.

[7] `https://pypi.org/project/azure-storage-file-datalake/`
[8] `https://pypi.org/project/azure-storage-blob/`
[9] `https://learn.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview`

Abstraction

To support access between the Factserver and multiple different blob types, the implementation introduces an abstract base class (`DataLake`). One of the main benefits of inheritance is code reuse, in this case storage-specific logic such as uploading, downloading and buffering. Child classes, such as `RawBlob`, inherit from this base and implement logic specific to their storage container. For example, the raw blob client defines its own file naming scheme but delegates upload mechanics to the parent class.
Another benefit of this design is robustness: With one single Data Lake class it would have been possible to invoke methods not meant for the currently instantiated client targeting a specific bucket. On the other hand, if one Data Lake class where to hold all clients for all buckets at the same time additional protection and duplication would have to be implemented to assign methods to the correct client.

Parquet

Apache Parquet[10] is a columnar storage file format optimized for analytical workloads. Its structure enables efficient compression and encoding, reducing storage costs and improving query performance especially when only specific columns need to be accessed. Parquet files store metadata in a file footer that includes schema definitions, column statistics and row group information, allowing efficient data skipping during reads. The format uses a hybrid encoding strategy and supports multiple compression algorithms to further reduce file size. In this project, Parquet was chosen for storage due to its suitability for large-scale data. Furthermore, Parquets compatibility with distributed processing frameworks like Apache Spark makes it a future-proof choice that can support advanced developments in upcoming projects.

Data Object Storage

To enable storage of the `data` dictionaries in Parquet, the Factserver transforms each `data` dictionary into a list of `predicate, object` pairs. This structure follows the Attribute Array pattern, as described in [16, 48]. The motivation behind this design choice lies in the fact that the schema of the `data` object varies significantly across Factsets. Using a nested dictionary format as was previously done in the `CouchDB` implementation (and discussed in section 3.1.3) would result in many possible columns in the Parquet schema and worsen query performance. The Attribute Array pattern allows for consistent schema definition and reduces the size of metadata parquet must compute and store ([33]). Furthermore, this approach provides flexibility for future extensions because the tuple can be easily expanded with additional fields for each column without requiring changes to the storage framework.
While this model increases flexibility, it may lead to slightly less efficient queries in scenarios where only a specific predicate is frequently accessed. Instead of using a top-level key lookup, the system must now filter rows where `predicate = 'birthPlace'`,

[10] `https://parquet.apache.org/`

which introduces an additional scan step.

4.4. Content Delivery Network

A CDN, as described in section 2.4 enables the system to fulfill the requirement of efficient information retrieval on a global level by caching the data most often requested. By caching frequently accessed data closer to end users, latency can be reduced, especially in geographically distributed environments. This decision also helps to offload traffic from the core infrastructure. This does improve the overall system robustness under high load. The selection of content to is critical in aligning the CDN with actual user behavior.
To implement the CDN component, Azure Front Door (AFD) was selected due to its tight integration with other Azure services. This choice aligns with the requirement of low-latency content delivery while keeping implementation overhead small. AFD provides security features such as Web Application Firewall and Distributed-Denial-of-Service protection to strengthen the reliability of the system.
While it is a common pattern in APIs that `GET` requests are cachable by default, `POST` requests are only cachable with certain headers explicitly allowing caching. `PUT` and `DELETE` are never cached and AFD only provides caching capabilities for `GET` requests as described in [32]. The Factserver API exposes some endpoints who fit this use case well, being often requested and providing information that gets updated rarely. These are:

Openapi API definition, only changes on Factserver update.

Precision Metric Retrieves information about different types or groups of metrics.

Domain List of domains the Factserver holds information on, only changes if a new source gets inserted into the system.

The configuration is as easy as selecting the endpoint to be cached in the route overview of the AFD. The complete set up is described in [35].
Factsets and statistics could be cached by `GET` request as well, however this requires the client to already be aware of the `id` to query for. This may be possible for some clients, for example Dashboard applications could hold their own list of `id`s in local storage and organize them around frequently requested topics. Even more changes would be necessary for `POST` endpoints, such as those which receive an entire new Factset in the request body to compute comparison. If the result is stored, as in the `compare/add` endpoint, the endpoint could be changed to return the comparison `id` instead of the result directly. The `id` could then be used in a cachable `GET` request. However, this caching only benefits the system if other users get the same `id` returned if they want to compare the same Factset, which is currently not the case. To achieve this a deterministic hash of the Factset could be created and used as an identifier. This shows that implementing such caching would require substantial change to the process.
To avoid unintended costs, AFD is not part of the implementation as the deployed resource would always listen for incoming requests to its configured URL. Since its setup is relatively straightforward, the configuration is considered trivial.

4.5. Comparison with Azure Reference Architecture

In [52] an Azure reference architecture (RA) for the FactCheck environment is proposed. Given that [52] is the main theoretical predecessor for this work, this section compares the implementation with the proposed architecture, highlights consistencies and discusses deviations.

4.5.1. Consistencies

1. The implementation uses ADF as proposed, however transformation tasks happen in collaboration with the Factserver.
2. The RA proposes a separation of the data between raw and cleaned datasets. This has been implemented with the 'cleaned' data in `CosmosDB` and the enriched bucket.
3. As proposed Azure Data Lake Gen2 is used as DL technology.
4. `CosmosDB` is used for cleaned data.
5. The data transformation follows the five-step structure proposed where ADF is triggered by file upload, the data is cleaned and transformed and then moved into the storage for end-user consumption.
6. The resources used are documented in Azure Resource Manager (ARM) templates.

4.5.2. Deviations

1. The RA proposes different resource groups for the Factserver and the database. While this could still be implemented in production, it is unpractical for development since the permission structure for sub-projects favors one resource group for each student.
2. It is proposed to archive raw data after expiration. This could still be implemented in production, but it is not granted for students as the archive storage tier is an offline-only storage with large overhead for access.
3. While `CosmosDB` is used for clean data, the JSON objects were separated into the DL to reduce the costs for storing and querying these large JSON objects.
4. In the RA the cloud-nativity of ADF is stressed. On the contrary, this work examines the possibility of self hosting the ADF environment to reduce costs.

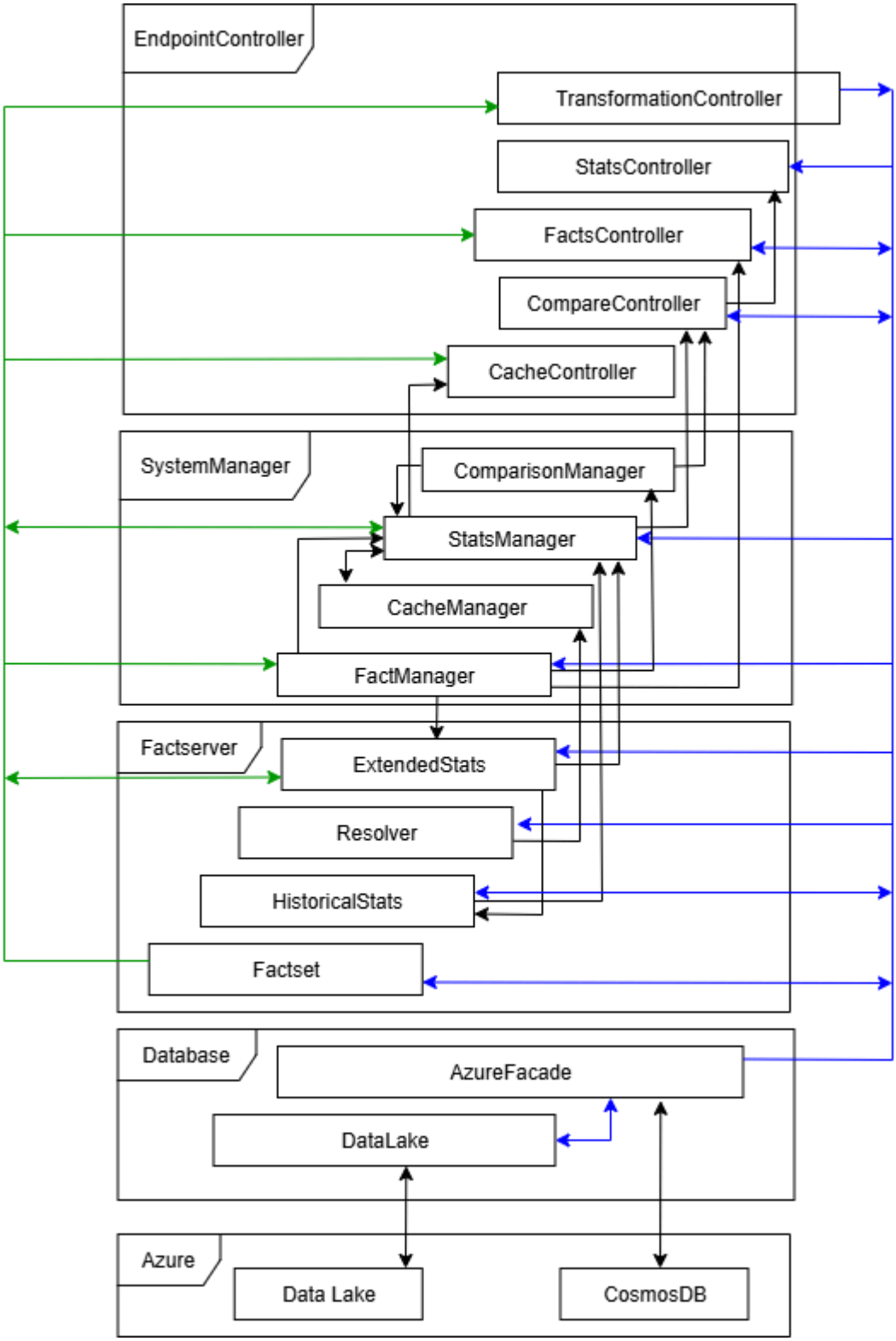

Figure 4.3.: Dataflow through the new Factserver

5. Implementation

This chapter covers the implementation of the cloud data storage framework designed in the previous chapter 4. The overall purpose of the adjustments to the existing Factserver is the replacement of the `CouchDB` specific code and enabling communication with the Azure-hosted storage solutions. For this reason changes have been made to a substantial amount of already present modules. Furthermore a database module has been introduced which holds several new classes concerned with the new azure components. The implementation for the new azure components is also described in this chapter. Finally, the Factservers API has also been extended with a suite of endpoints that enable data transformation towards azure-specific formats.

5.1. Data Factory Pipeline

The processing from raw crawler data of scraped web pages towards the format used by the Factserver in this new data storage framework is done in an Azure Data Factory pipeline. The pipeline is structured into two halves, there the first part ingests a newly uploaded document and the second part iterates over each JSON object found inside that document.

5.1.1. Limitations

As described in section 4.3.1, to save costs the pipeline is designed to support execution on a SHIR. One downside of this approach is that the 'Data flow' activity type, which can be used for transformation operations like renaming and merging columns, is only supported on Azure IR [25]. Since the Factserver and with it the Azure Functions for transformation are run in a Docker container and not in the Azure environment, 'Web' activities are used for communication sending the information to preset URLs (example shown in figure 5.1). If the Azure Functions were deployed directly into the Azure environment, they could be accessed by the 'Azure Function' activity type by a specified name. While 'Web' activities used to time-out after one minute of waiting for a result [41], since this behavior has been changed, no difference in execution between the two activity types is left. One note to this is that during the setup of this implementation the connection to Azure Functions was tested, but the Data Factory had issues finding the deployed functions, even though they were accessible through standard web interfaces like the browser or Postman and a key for authentication had been provided to the factory. For the reasons described this issue was not further investigated.
One final limitation applies; the pipeline uses a for-each loop to apply changes to every dictionary in the input file. During this, variables such as `type`, `id` and `name` are set using

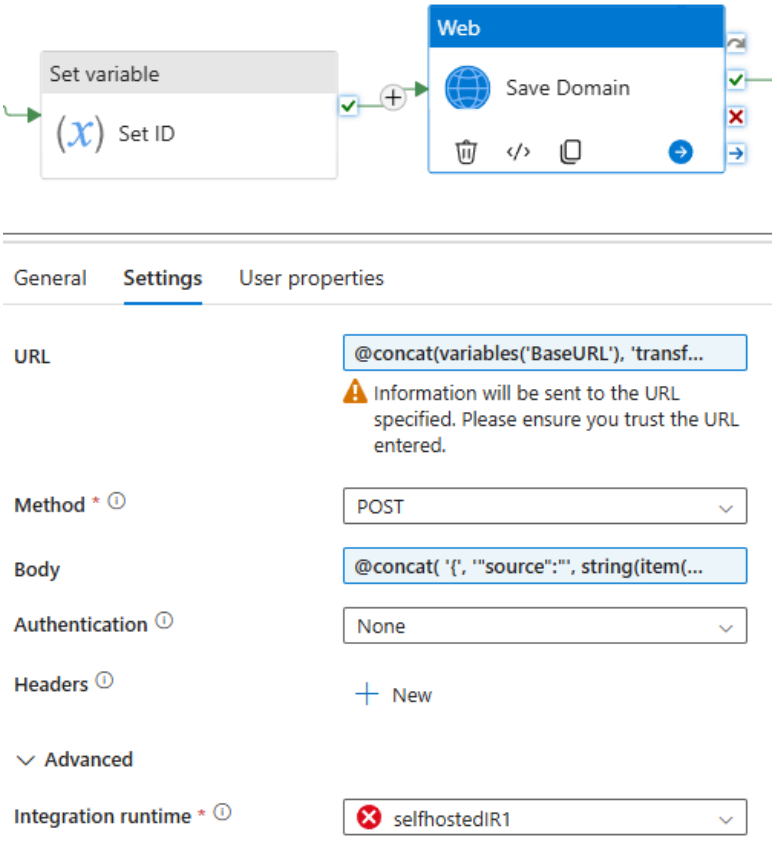

Figure 5.1.: Settings of the Web activity to save the domain

the 'Set Variable' activity. As described in [38], these variables are not limited to the scope of the iteration. For this reason, execution in parallel is not possible, since the assignment of the variables could interfere with other iterations of the for-each loop. Currently, with only one instance of the Factserver running in production, this is not an issue, however the implications of this would need to be considered if the entire architecture would be replicated for parallel execution as-is.

5.1.2. Pipeline Trigger

The pipeline trigger is a Blob Events Trigger of type `BlobEventsTrigger`. It monitors the staging folder in the raw bucket in the DL. The purpose is to start the pipeline automatically whenever a new text file arrives. For this reason the trigger listens for the event `Microsoft.Storage.BlobCreated`, indicating the creation of a new blob. If a new pipeline is started, the name of the created files need to be communicated into the pipeline to enable the first activity to retrieve those files. For this purpose, the pipeline parameter `triggerSourceFolder` and `triggerSourceFile` are provided with information from the pipeline body: `@triggerBody().folderPath` and `@triggerBody().fileName`.

Enabling this trigger creates an event-driven design which creates new Factsets and calculates their statistics in near-real-time. In the `CouchDB` data storage framework the statistics would be calculated only once every day, causing inconvenience to developers and users.

5.1.3. Pipeline Activities

The pipeline consists of 17 activities. In the following list indentation is used to highlight that activities reside in another activity:

RawJsonLookup retrieves the files based on the trigger information

Set FactServer Constant sets the base URL of the FactServer to quickly switch for example from `localhost` to the production URL by changing the string stored in the activity

Set Data Array cuts the text file, which consists of one JSON document per line, into an array of JSONs each representing one `FactSet`

ForEach Factset one iteration handles one JSON

- **Set Name** stores the name of the `FactSet` as a pipeline variable
- **Set Name to unknown** if no name is found, the name variable is set to 'unknown'
- **Set Type** stores the type of the `FactSet` as a pipeline variable
- **Set ID** generates a unique ID for the `FactSet` using the built-in function `@guid()`
- **Save Domain** calls the domain endpoint described in section 5.5.3
- **Check MID** If condition; checks if MID is already provided
  - **Set MID from item** If-Case: MID is already provided in the JSON
  - **query Google KnowledgeGraph** calls the resolve endpoint described in section 5.5.4 if no MID is given
  - **Set MID** sets variable from query result
  - **MID Error** optionally calls the error endpoint described in section 5.5.6
- **Sink to CosmosDB** calls the `CosmosDB` endpoint described in section 5.5.2
- **Sink Data to DL** calls the data endpoint described in section 5.5.1
- **Calculate Stats** calls the update endpoint described in section 5.5.5

At the end of the pipeline the file could have been removed again using a 'Delete' activity, however this was not implemented, focusing on traceability and debugging instead of keeping the `raw` bucket as small as possible.

5.2. FactServer: DataLake

To interact with the Azure Data Lake storage described in section 4.3.4, the system implements an abstract base class `DataLake` on top of Azure's `DataLakeServiceClient`. It encapsulates functionality such as file upload, download and DataFrame appending while limiting the need for child-classes to interact with Azures Software-Development-Kit themselves. Mocking and caching is enabled through an emulation in the local file system. The children share a simple cache using the Least-Recently-Used strategy. This helps prevent unnecessary repeated downloads for large Parquet files. The structure of the module can be seen in 5.2.

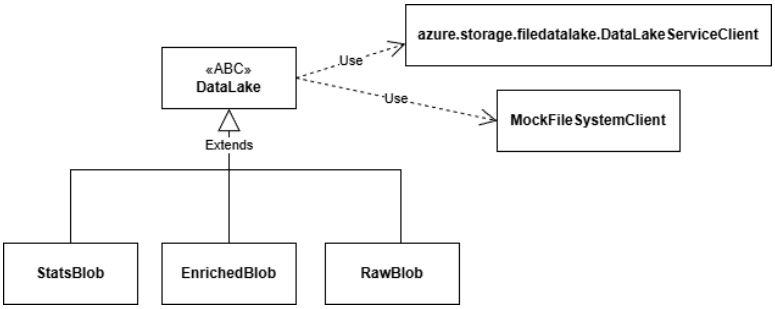

Figure 5.2.: Class diagram for the DataLake module

5.2.1. Data Ingestion Basics

File Upload

File uploads in the Factserver can be triggered at any time using the API, however most uploads are expected to be triggered from the pipeline described in section 5.1. File uploads in the DL are designed to fully overwrite existing files as shown in listing 5.1. Appending must be handled explicitly upstream. While the DL class supports uploads of any file type, this decision was driven by the importance of Parquet for the implementation. While Azure's API allows appending bytes to a file, the Parquet format itself is not designed to support appending. Valid appends would require rewriting Parquet rows and metadata, which introduces complexity and risks corrupting the file. Therefore, the system always reads the existing file into memory, merges it with new data using Pandas[1], and overwrites the file in one operation. While not a driver for this design decision, this procedure also ensures written rows are immutable which can help produce predictable behavior in applications querying the DL files. Still developers might choose to extend this implementation with custom wrappers to break the immutability if it fits their use case.

```
1 file_client = directory_client.create_file(file_name)
2 file_client.append_data(data=content, offset=0, length=len(content))
```

[1] https://pandas.pydata.org/

```
file_client.flush_data(len(content))
```

Listing 5.1: Overwrite-only upload of a file buffer

Handling Schema Drift in Parquet Files

One challenge when storing semi-structured data is schema drift where the structure of incoming data can vary over time or between projects. In the context of FactCheck this is especially important as different sub-projects may include different statistic or metadata columns. To address this, the system relies on `pandas.concat()` to merge incoming and existing `Dataframes` as shown in 5.2. When new columns are introduced, Pandas automatically fills missing values with `NaN`, ensuring seamless compatibility.

```
existing_df = pd.read_parquet(buffer_old)
df = pd.concat([existing_df, new_df], ignore_index=True)
```

Listing 5.2: Concatenating dataframes with flexible schemas

This allows the DL to handle evolving schemas in Parquet files implicitly while writing these append-only files.

Mocking Data Lake Service Client for Testing

While Microsoft provides the Azurite emulator to enable local testing of `BlobStorageServiceClient`, no such service exists for the `DataLakeServiceClient`. To avoid unnecessary load and reduce costs during testing, the system can use a mocked file system client when executed in local or testing environments as displayed in 5.3. This is useful for tests involving repeated uploads as they are done during automated testing with `Schemathesis`. The Dependency Injection of the mock client is triggered when `localhost` is provided as the value for the DL account name in the configuration file. For storage the `tempfile`[2] library is used to return the location of a temporary directory.

```
if account_name == "localhost":
    self.file_system_client = MockFileSystemClient(file_system_name=
    tempfile.gettempdir())
else:
    self.account_url = f"https://{account_name}.dfs.core.windows.net"
    self.service_client = DataLakeServiceClient(account_url=self.
    account_url, credential=account_key)
```

Listing 5.3: Mock client fallback for test execution

Another functionality is testability while avoiding interaction with production systems and data.
The `MockFileSystemClient` acts as the main entry point and can create `FileClient`, `DirectoryClient` and `StreamDownloader` objects. These clients write actual files within the system's temp directory. When a mock file is flushed, its binary content is written using standard I/O operations as shown in listing 5.4.

[2] `https://docs.python.org/3/library/tempfile.html`

```
with open(self.file_path, "wb") as f:
    f.write(self._write_buffer)
```

Listing 5.4: Flushing data to local mock file

During reads, file system specific errors are replaced by the adequate error that would be thrown by Azure as highlighted in5.5.

```
def readall(self) -> bytes:
    try:
        self.logger.debug("Opening file at : %s", self.file_path)
        with open(self.file_path, "rb") as f:
            return f.read()
    except FileNotFoundError:
        raise ResourceNotFoundError(f"Mock file not found:  {self.
    file_path}")
```

Listing 5.5: Replacing file system error

5.2.2. Raw Storage Bucket

For storing raw, unprocessed data, the system defines a small subclass of the `DataLake` abstraction. This functionality was implemented to continue support of API endpoints that would upload `FactSet`s to the CouchDB. This subclass, `RawBlob`, writes incoming JSON payloads to the `raw` container in Azure DL as can be seen in listing5.6.

```
class RawBlob(DataLake):
def upload_file(self, serialized_json):
    buffer = BytesIO(serialized_json.encode("utf-8"))
    self.upload_buffer(buffer, f"staging/upload_from_factserver-{datetime
    .now().strftime('%Y%m%d-%H%M%S')}.txt")
```

Listing 5.6: Uploading raw JSON with timestamped filename

The timestamp is appended to the filename to ensure uniqueness as well as traceability between the Factserver activity and DL results. The data is not parsed or validated at this stage but can be picked up by the event-based trigger of the Data Factory described in section 4.3.1.

5.2.3. Enriched Storage Bucket

The `EnrichedBlob` encapsulates access to structured and cleaned Factset data, stored in Parquet format within a hierarchical structure using `fact_type` and `fact_name` as folder and file name. The actual structure of the stored parquets can be adapted to suite the development of new use-cases. Optionally, a `fact_id` can be provided to filter rows within a file as shown in listing 5.7. The combined storage of similar dictionaries in one file enables efficient Parquet compressing, which would be less effective if Parquet files were created more granular per ID.

```
df = pd.read_parquet(buffer)
if fact_id is not None:
```

```
    df = df[df["fact_id"] == fact_id]
return dict(zip(df["predicate"], df["object"]))
```

Listing 5.7: Selective access to factset data

The final return type is a key-value dictionary that mirrors the legacy format used with CouchDB. If the file does not exist the method returns `None`, leaving the handling of missing content to downstream methods based on their requirements and expected behavior.
Structured data is written back to the DL using the inherited append mechanism.

5.2.4. Statistics Storage Bucket

The `StatsBlob` subclass is responsible for storing statical data on the Factsets. Currently there are three major types of statistics in the FactServer: domain-level, factset-level and historical statistics. Due to difference in schema and use, every type is handled differently in the implementation.

Domains

One functionality of the `StatsBlob` is storing a list of source domains in the domain text file. Example content of this file is provided in the appendix A.2. To support fast lookup of known domains the domain list is loaded into a Python `set` as described in listing 5.8.

```
buffer = BytesIO()
self.download_file("domain/domain.txt", buffer)
list_content = set(line.strip() for line in buffer.read().decode("utf-8")
    .splitlines())
```

Listing 5.8: Loading domain list into memory

To avoid duplication on upload, first it is checked if the domain is already present in storage as shown in listing 5.9.

```
if domain not in domains_lst:
    combined = domains_raw + f"{domain}\n"
    self.upload_buffer(BytesIO(combined.encode("utf-8")), "domain/domain.
    txt")
```

Listing 5.9: Appending a new domain entry if needed

While this approach is not optimized for large-scale operations it works well for this use case because the list of domains is small and changes infrequently. `.txt` is a simple format that supports fast lookups and append-only writes on this single list of domains.

Factset Statistics

The `StatsBlob` class supports storage and retrieval of Factset statistics which used to be stored in the `stats` dictionary in `CouchDB`. Like the data in the enriched blob, Parquet files are organized by `fact_type` and `fact_name` and contain multiple records associated with a `fact_id`. Opposed to the data in the enriched blob, each statistic for a Factset is

organized in a single row as shown in listing 5.10 because there is a one-to-one relationship between statistics and `FactSet` and there is no need for dynamic separation of the data into multiple rows. Statistics are described in a more rigid schema which is not expected to change within each upstream use case but might do so between use cases in different projects in the FactCheck context.

```
new_factstat_stats["id"] = fact_id
df = pd.DataFrame([new_factstat_stats])
self.append_df_to_dl_parquet(df, path)
```

Listing 5.10: Appending new statistic row to parquet

To avoid issues with `NaNs` inserted by Pandas during schema merging, the DataFrame is normalized before conversion to a list of dictionaries as can be seen in listing 5.11.

```
df = df.where(pd.notnull(df), None)
return df.to_dict(orient="records")
```

Listing 5.11: Normalizing NaN to None for JSON export

Historical Statistics

To store historical statistics, Parquet files are partitioned by date with each row containing a `domain`, `date` and a set of metrics as shown in listing 5.12.

```
hist_stats["domain"] = domain
hist_stats["date"] = date
df = pd.DataFrame([hist_stats])
self.append_df_to_dl_parquet(df, path)
```

Listing 5.12: Appending a historical stats entry

This enables the `StatsBlob` class to efficiently query for time ranges using the file names in the `historical_stats` directory as can be seen in listing 5.13.

```
available_files = self.file_system_client.get_paths(path="
    historical_stats")
        available_dates = {
            path.name.split("/")[-1].replace(".parquet", ""): path.name
            for path in available_files
        }
        ...
    if date_str in available_dates:
        tmp_buffer = BytesIO()
        self.download_file(f"historical_stats/{date_str}.parquet",
    tmp_buffer)
```

Listing 5.13: Filtering available Parquet files by date

After this, the method filters for the requested domain and concatenates all matching rows into a single output list of date-value pairs, as it would have been produced by `CouchDB` as highlighted in listing 5.14.

```
return [[row.pop('date'), row] for row in hist_dict]
```

Listing 5.14: Formatting as date-value pairs

fact_stats Dictionary

In the legacy `CouchDB` format, `fact_stats` was nested as a dictionary within the main statistics object. Since Parquet does not support arbitrarily nested JSON-like structures, the system flattens the content into a table before upload by storing the name of the metric in a dedicated column as shown in listing 5.15.

```
flattened_rows = []
for name, stats in fact_stats.items():
    stats["id"] = fact_id
    stats["name"] = name
    flattened_rows.append(stats)
```

Listing 5.15: Flattening nested fact_stats dict

The transformed statistics are then stored in the same type, name and id format as the other Parquet files described.
To maintain compatibility with consumers expecting a nested `fact_stats` dictionary, the system re-hydrates the flat table into its original structure after reading. This includes converting `NumPy`[3] arrays, which have been created by `Pandas` during conversion, back to JSON-compatible lists as can be seen in listing 5.16.

```
def convert_inner_parque(self, val):
    if isinstance(val, np.ndarray):
        return val.tolist()
    ...
transformed_fact_stats[name] = {
    k: self.convert_inner_parque(v) for k, v in fact_stat_dict.items() if
     k not in ("name", "id")
}
```

Listing 5.16: Rehydrating flat rows into nested dictionary

The alternative to this approach would have been to store the dictionary as one string in a single column. While this would reduce the complexity of transformation there would also have been retrieval and storage performance penalties.

5.3. FactServer: Azure Facade

The `AzureFacade` class handles all connections with Azure, coordinating access to the DL using the `DataLake` class discussed in the previous section while also querying `CosmosDB`. It's design was described in section 4.3.2.

5.3.1. Abstract Database Interface

To maintain compatibility with the current `CouchDB`-based logic an interface is defined which the `AzureFacade` must implement. This abstraction ensures that the same functionality is exposed as in the legacy system.

[3] `https://numpy.org/`

```python
class DatabaseInterface(ABC):
    @abstractmethod
    def get_object_by_id(self, id: str): pass
    @abstractmethod
    def query_view(self, view, limit=None, key=None, ...): pass
    @abstractmethod
    def save_object(self, obj): pass
    @abstractmethod
    def delete_object(self, obj): pass
```

Listing 5.17: Database interface for backend compatibility

5.3.2. Initialization

The `AzureFacade` constructor reads configuration data, sets up clients and validates connections to ensure the system is operational. To enforce singleton behavior, a module-level flag is checked during initialization. If an instance has already been created, a warning is logged, and re-initialization is skipped as can be seen in listing 5.18.

```python
if AzureFacade.__initialized is True:
    self.logger.warning("Warning: AzureFacade instance already exists")
```

Listing 5.18: Singleton check in AzureFacade constructor

Database credentials and parameters are loaded from a configuration file. The constructor initializes the `CosmosDB` client, ensures that the specified database and container exist and stores the credentials in memory for later use. If the database or container do not exist, they will be created as highlighted in listing 5.19.

```python
self.client = CosmosClient(config['uri'], config['key'])
self.db = self.client.create_database_if_not_exists(id=config['db'],
    offer_throughput=400)
self.db.create_container_if_not_exists(
    id=config['container'],
    partition_key=PartitionKey(path="/mid"),
)
```

Listing 5.19: Creating database and container if not exists

To avoid initializing all service clients upfront the system uses the `@cached_property` decorator as shown in listing 5.20. This pattern ensures that the object is constructed only on first use but cached for all further access.

```python
@cached_property
def _raw_blob(self):
    return RawBlob(self.dl_account_name, self.dl_account_key)
```

Listing 5.20: Lazy instantiation of blob clients

The same pattern is used for `CosmosDB`, enriched and statistical blob clients. These clients are perfectly suited for lazy loading and caching because their initialization is expensive, but the objects never change once they are created.

5.3.3. Delegation to Data Lake

The `AzureFacade` class includes a set of wrapper methods that delegate DL operations to the appropriate blob client classes: `RawBlob`, `EnrichedBlob` and `StatsBlob`.

RawBlob Access

- `upload_to_raw_bucket(serialized_json)`

EnrichedBlob Access

- `get_data(fact_id, fact_type, fact_name)`
- `upload_data_parquet(df, path)`

StatsBlob Access

- `get_domains()`
- `insert_domain(domain)`
- `get_stats(fact_id, fact_type, fact_name)`
- `upload_stats(fact_id, fact_type, fact_name, new_stats)`
- `get_extended_stats(fact_id, fact_type, fact_name)`
- `upload_extended_stats(...)`
- `save_historical_stats(domain, date, hist_stats)`
- `get_historical_stats(domain, start, end)`

For stats and `extended_stats` methods the Facade share the same method exposed by the `StatsBlob` and discussed in section 5.2.4, they just add different postfixes to the paths to differentiate between the two types of Parquets. Additionally, the `extended_stats` methods also handle the combination with the `fact_stats` dict described in section 5.2.4, hiding this extra processing from application in front of the facade as shown in listing 5.21.

```
1 e_stats = self._stats_blob.get_stats(fact_type, fact_name + '_extended',
      fact_id)
2         if e_stats is not None:
3             e_stat = e_stats[-1]
4             e_stat["fact_stats"] = self._stats_blob.get_fact_stats(
      fact_id=fact_id,
5                                                 fact_type=fact_type,
6                                                 fact_name=fact_name)
7             return e_stat
```

Listing 5.21: Loading fact_stats into extended_stats

The last dictionary from the list is taken using Pythons -1 operator. While a list of length 1 would be expected, and currently no process that could circumvent this exists in the Factserver, this secures against the possibility that a calculate stats method was called twice on the same Factset, resulting in multiple rows in the Parquet file associated with one Factset.
While the blob classes could be used directly by other modules, access through the facade helps with separation of concern while making the code easier to understand and extend.

5.3.4. CosmosDB Access

Two different approaches for access to CosmosDB data were included in the implementation.

Retrieval by Attributes

The method `_get_object_by_attribute` enables database lookup using a combination of fields. When both the object ID and its partition key (MID) are provided, it automatically uses `CosmosDB's` optimized `read_item()` method without requiring additional developer input as can be seen in listing 5.22.

```
if "id" in attr_values and "mid" in attr_values:
    item = self._container.read_item(item=attr_values.get("id"),
    partition_key=attr_values.get("mid"))
```

Listing 5.22: Optimized lookup using ID and partition key

If other fields are provided, the method constructs a SQL query by iterating other the provided dictionary of attributes and value pairs, placing the name of the requested value in the SELECT part of the query and adding the value into the WHERE clause as highlighted in listing 5.23. Offset and limit can also be specified to reduce load on the database, similar to what the Factserver would provide for users. An example of what the query could look like is provided in the appendix A.3.

```
select_clause = "Select "
            if not include_metadata:
                select_clause = select_clause + COSMOS_COLUMNS
            else:
                select_clause = select_clause + "* "
where_conditions = []
params = []
for attr, value in attr_values.items():
    where_conditions.append(f"c.{attr} = @{attr}")
    params.append({"name": f"@{attr}", "value": value})

where_clause = " AND ".join(where_conditions)
query = f"{select_clause} FROM c WHERE {where_clause} "
query += f"OFFSET {skip} LIMIT {limit}"
```

Listing 5.23: Dynamic filter query with parameters

This private method enables reuse across many higher-level methods. It will also remove metadata unless specifically requested as shown in listing 5.24.

```
for c in ["_rid", "_self", "_etag", "_attachments", "_ts"]:
    item.pop(c)
```

Listing 5.24: Removing internal metadata fields

To simplify use of this dynamic method, the `AzureFacade` provides a set of thin wrapper methods that make the usage of the facade more readable and simplify testing by providing clear points for mock injection.

Wrapper Method Examples

- `get_object_by_id(id, mid=None)`
- `get_object_by_mid(mid)`
- `get_object_by_source(source)`
- `get_object_by_source_mid(source, mid)`
- `get_object_by_source_name(source, name)`
- `get_object_by_name_type(name, type)`
- `get_object_by_name_mid(name, mid)`

`get_object_by_id` will return a single dictionary since id is guaranteed to be unique. All the other wrappers will produce a list of dictionaries.
For use cases requiring more complex data retrieval, developers can also send custom queries to the facade using the `execute_query` method which will delegate the provided query to `CosmosDB`.
The method `save_object` is responsible for inserting new Factset entries into `CosmosDB`. Unlike the legacy `CouchDB` implementation, `CosmosDB` requires every document to include both an ID and a partition key. These should be generated upstream, for example in the ingestion pipeline. To ensure schema consistency, the method also validates that all values are strings as shown in listing 5.25.

```
for key, value in obj_dict.items():
    if not isinstance(value, str):
        raise ValueError(...)
```

Listing 5.25: Validating that object contains only string values

This prevents nested dictionaries from being inserted which could be a classic `CosmosDB` use case but is delegated to the DL for the Fact-Check context.

Retrieval by Emulating CouchDB Views

The method `query_view` simulates the concept of `CouchDB` views by mapping view names to dynamically generated `CosmosDB` SQL queries. Optional postprocessing and reduction steps allow the system to emulate CouchDB's map-reduce capabilities. To maintain compatibility with the legacy implementation the method also supports the parameters optional in CouchDBs `query_view` method `key`, `start_key`, `end_key`, `include_docs` and `reduce`. The only parameter skipped is `group` because it was not used in the FactServer. The method contains an if/else statement which constructs the queries and methods to simulate the requested view. Covered views are:

- `basic/all`
- `basic/domain`
- `basic/mid`
- `basic/mid_grouping`
- `basic/source`
- `name_type/nomid`
- `source_name/mid`
- `stats_domain_date/r-stats`

Depending on the view selected the if/else statement will assign query columns, a `WHERE` clause, a postprocessing function and a reduction function. After the if/else statement the operations are performed in the following order to simulate the behavior:

1. Query `CosmosDB` using the query, where clause, skip and limit provided
2. Transform the result using the postprocessing (map) function
3. If given, filter the transformed data using the value provided as the key parameter (as shown in listing 5.26)
4. If given, apply a range filter on the transformed data using the provided start and end key (as shown in listing 5.27)
5. Use the reduction function on the filtered data (as shown in listing 5.28)
6. Return the result

```
1 if key is not None:
2     results = [item for item in results if item.get("key") == key]
```

Listing 5.26: Filter results by exact key match

```
if start_key or end_key:
    results = [item for item in results if
               in_range(start_key=start_key, end_key=end_key, range_key=
    item.get("key"))]
```

Listing 5.27: Filter results by key range (start and end)

```
if reduce:
    values = [row['value'] for row in results]
    reduced_values = reduce_processing(values)
    results = [{"key": None, "value": reduced_values}]
```

Listing 5.28: Reduce result set using provided reduction function

While the range filter generally follows basic Python syntax, one `CouchDB` quirk has to be accounted for, which is the use of empty dictionaries in composite keys to specify an unlimited upper range. For this custom behavior is implemented that can be seen in listing 5.29.

```
def _compare_keys(a, b):
    for x, y in zip(a, b):
        if isinstance(x, dict):
            return 1
```

Listing 5.29: Force dictionaries to sort highest in range filter

In listing 5.30 the processing of `CosmosDB` results for the basic/source view is shown as an example. The method has two parts: First, by iterating over the `CosmosDB` result, the unique sources are collected and counted. In the second step the processed information is formatted using a template recreating the output from the `CouchDB` view with key and value fields in a dictionary.

```
def source_processing(source_pre_results):
    """ for basic/source"""
    source_counter = Counter()
    for item in source_pre_results:
        source = item.get("source")
        if source:
            source_counter[source] += 1
    source_results = [
        {"key": source, "value": count}
        for source, count in source_counter.items()
    ]
    return source_results
```

Listing 5.30: Processing CosmosDB results to simulate CouchDB behavior

In listing 5.31 the reduction of already processed `CosmosDB` results for the basic/mid_grouping is shown as an example. The method is optional; therefore, the input it will be provided is already in a format which would also be ready to be returned from the `query_view` method and is thus in simulated `CouchDB` format. In contrast to the mapping function in listing 5.30, this reduction function returns a single dictionary with a list of types and a single integer for the count key. An example output is provided in the appendix A.3.

```python
def reduce_mids(post_results):
    """reduction for basic/mid_grouping"""
    types = set()
    count = 0
    for item in post_results:
        if "type" in item:
            types.add(item["type"])
            count += 1
    summary = {
        "types": list(types),
        "count": count
    }
    return summary
```

Listing 5.31: Reducing processed results to simulate CouchDB behavior

A final example for mapping is highlighted in listing 5.32 which showcases the function retrieving and working with statistics from the DL to simulate the stats_domain_date/r-stats view.

```python
for item in stats_pre_results:
    r = item['result']
    stats = self.get_stats(r['id'], r['type'], r['name'])
    if stats is None:
        continue
    stats["source"] = r["source"]
    stats["name"] = r["name"]
    try:
        domain = r["source"].split("/")[2]
    except IndexError:
        domain = None
    stats_results.append({
        "key": [domain, r["date"]],
        "id": stats.pop("id"),
        "value": stats
    })
```

Listing 5.32: Querying the Data Lake for stats inside of query_view

5.4. FactServer: Internal Factset Representation

The `Factset` class represents a single unit of processed data extracted from a website. It holds metadata, facts and sets of statistics and acts as a core entity in the Factserver, for example in processes calculating comparison results or during the retrieval of JSON documents via the FactServers Web API. Changes made to the schema of the `Factset` for the transition to the new architecture are described in section 4.2.2. Additionally to the changes in the schemas various updates had to be made to the `Factset`s logic and methods. Its design uses the `@dataclass`[4] decorator to simplify boilerplate code such as initialization. The constructor intentionally excludes direct setting of complex fields

[4] `https://docs.python.org/3/library/dataclasses.html`

like `data`, `stats` and `extended_stats`, which are instead lazily loaded from Azure and internally managed via private attributes.
To improve performance and flexibility, multiple class constructors are defined:

- `from_id()`: Retrieves the object by querying `CosmosDB` by ID. The result is directly unpacked into the `dataclass` constructor, with exceptions caught and re-thrown as domain-specific errors `FactsetValidationException` and `NoResultsFoundError`.
- `from_id_and_mid()`: Like `from_id()`, but optimized for performance by including the `mid` as a partition key in the query. This reduces query time and cost as described in section 5.3.4.
- `from_json()`: Accepts a Factset fully prepared as JSON dictionary. If a `data` field is present, it is temporarily removed before instantiation to avoid breaking the constructor's schema and instead injected via a dedicated setter.

The class also implements a `to_dict()` method, which calls all lazily loaded fields prior to serialization and is used for example when Factsets are printed on API requests or for debugging. For this purpose, the legacy implementation already held a custom `JSONEncoder` class which was called in the `jsonout` function. Therefore, this default method had to be extended with `to_dict()` as shown in listing 5.33

```
def default(self, o):
    if isinstance(o, Factset):
        return o.to_dict()
    return o.__dict__
```

Listing 5.33: Factset to JSON custom behavior

When `Factset`s are needed in the FactServer, they are loaded from disk by combining the related information from the various specialized storage solutions. As previously mentioned, the `Factset` class uses manual lazy loading for its optional fields `data`, `stats` and `extended_stats` to avoid unnecessary calls to the DL. Instead of using Python's `@cached_property` decorator—which caused unpredictable behavior when used together with the `@dataclass` decorator the class uses explicit getter methods such as `get_data()`, `get_stats()` and `get_extended_stats()` as shown in listing 5.34.

```
def get_data(self) -> dict:
    """lazy loading"""
    if self._data is None:
        self._data = azure_facade.get_data(self.type, self.name, self.id)
    return self._data
```

Listing 5.34: Lazy loading private data dictionary

The setters not only write updated dictionaries to the DL but also invalidate the lazily loaded fields by resetting the value to `None`. This ensures that read operations always retrieve the latest version from Azure without requiring a new `Factset` object. These new getters and their critical logic for information retrieval have the consequence that the use of `factset.data` to access the data attribute must be refactored throughout

the Factserver. Developers planning on extending the implementation mustn't access the attribute directly to avoid unexpected behavior. The attributes have been renamed following the Python naming convention for private attributes with a leading underscore, however Python does not restrict access to private attributes.
The hash and equality operators are now created solely by comparing the unique id of the `Factset`.
In comparison to the legacy `Factset` class a lot of validation has been removed. Instead, the load of ensuring data validity has been moved to the dedicated data transformation pipeline described in section 5.1. While the option to create Factsets directly from JSON is given for flexibility, it is not the best practice approach to working with the `Factset` class.
Figure 5.3 shows the sequence of retrieving a Factset with all associated information. For this example, the use case of returning a Factset by `id` for the `FactController` endpoint was used.

5.5. Transformation Controller

The transformation controller exposes several endpoints implemented as Azure Functions. They are designed to be the interface to the data transformation pipeline in Azure Data Factory, the design for which is described in section 4.3.1. The pipeline calls the endpoints in the order they are listed in in this section. In doing so the raw information gets transformed into the enriched Factset where information is split according to the data storage framework. In accordance with Azure Functions best practice, all the endpoints are stateless. All endpoints return status code 500 on unexpected failure.

5.5.1. Endpoint: transformation/data

This endpoint serves to convert the semi-structured data dictionaries in raw Factset data into a normalized format that is fit for storage in Parquet files. As described in section 4.3.4, the Attribute-Array pattern was chosen for this format.
The endpoint accepts a PUT request containing a JSON body with four mandatory fields: `id`, `type`, `name` and `data`. The transformation logic is shown in listing 5.35. Null values and metadata starting with an underscore are filtered out, while dictionaries and lists are converted to strings.

```
1  def transform(data, fact_id):
2      new_data = []
3      for key, value in data.items():
4          if value is None or key.startswith(""):
5          continue
6      new_data.append({
7      "fact_id": fact_id,
8      "predicate": key,
9      "object": json.dumps(value) if isinstance(value, (dict, list)) else
       value
10     })
```

```
11      return new_data
```

Listing 5.35: Transformation of data dictionary into attribute array

Each key value pair in the `data` dictionary is associated with the id of the Factset. This enables storing data for multiple Factsets in the same Parquet (See also section 5.2.3). After the transformation the `azure_facade` is used to pass the data into the enriched bucket.
This endpoint will return status code 200 on successful upload or 400 if converting the data failed because of malformed JSON, missing keys or data of the wrong type.

5.5.2. Endpoint: transformation/cosmosdb

The `transformation/cosmosdb` endpoint is a minimal wrapper delegating the POST request body into the `azure_facade` for storage in `CosmosDB` using the `save_object()` method described in section 5.3.4. In the context of the data transformation pipeline the expected input would be the attributes of the `FactSet`.
It will return status code 201 if the upload to `CosmosDB` worked and 400 if the JSON was malformed or held unexpected types such as nested dictionaries. Finally, 409 will be returned if an object with the provided id already exists.

5.5.3. Endpoint: transformation/domain

The `transformation/domain` endpoint extracts the domain name from the `source` provided in the body (expected to be a URL) and appends it to the `domain.txt` file in the DL using
`insert_domain()` as described in section 5.2.4. The domain is expected to be the third string in a /-separated `source` URI.
The endpoint will return status code 201 on successful storage of the domain and 400 if the JSON was malformed. If the endpoint cannot find a domain in the `source`, it will return a status code 200 indicating that no action is taken.

5.5.4. Endpoint: transformation/resolve

The `transformation/resolve` endpoint delegates the retrieval of a machine identifier (MID) to the `Resolver` class. To fulfill the format expected by `Resolver.resolve()`, the endpoint reads `source`, `name` and `type` parameters from the GET request and wraps them in a minimal Factset-like dictionary as can be seen in listing 5.36.

```
1       factset_json = {
2           "source": req.params.get("source"),
3           "data": {
4               "name": req.params.get("name"),
5               "@type": req.params.get("type")
6           }
7       }
8       mid = Resolver().resolve(factset_json)
```

```
9     return HttpResponse(mid, status_code=200)
```

Listing 5.36: Resolve MID via Google Knowledge Graph

If the resolver fails to find a find a MID a 204 no content code is returned. On successful retrieval the resolved MID is sent back with a 200 status code. As with the previous endpoint a 400 is returned if the parameters cannot be parsed correctly.

5.5.5. Endpoint: transformation/update

The `transformation/update` endpoint initiates the calculation and storage of statistics for one specific Factset which is specified using the `id` and `mid` provided in the POST request body. The information from the Factset needed for the calculation of the statistics is collected using the new best-practice of calling the appropriate constructor, `Factset.from_id_and_mid()` (described in section 5.4) and leaving the retrieval to the `azure_facade`.
Successful calculation return status code 200. If not Factset can be found with the provided information a 204 is returned. Invalid input results in a returned 400.

5.5.6. Endpoint: transformation/error

Finally, the `transformation/error` endpoint simply takes the provided request body as JSON and logs it using the Factserver's loggers. This endpoint is designed to allow the data transformation pipeline to propagate errors outside of the Factserver into the Factserver's logs.

5.6. Migration

To facilitate automated migration from `CouchDB` to Azure a script is provided in `utils/migration.py` that handles all data transformation internally in the Factserver to save the cost associated with moving this large dataset over the internet multiple times. It could be started for example using the command-line.
To ensure robustness and fault tolerance during the migration process, a simple checkpointing mechanism is implemented using a single-line text file named `startkey.txt` as described in listing 5.37. This file stores the last document ID in the latest processed chunk and enables the migration script to resume from this exact point in case of failure or manual termination. This is possible due to the order provided by the `CouchDB _all_docs` view.

```
1 def read_startkey_from_file_if_exists():
2     with open(file_path, "r", encoding="utf-8") as f:
3         line = f.readline().strip()
4         return line if line else None
5 def write_startkey_to_file(startkey: str):
6     with open(file_path, "w", encoding="utf-8") as f:
7         f.write(startkey + "\n")
```

Listing 5.37: Checkpoint for migration

During the migration the script retrieves documents using `CouchDB's` Python API and transforms them into the required schema for `CosmosDB` and the DL. Each iteration handles a configurable number of documents as can be seen in listing 5.38. For this purpose, `iterations_` and `chunk_size` can be provided as input parameters.

```
for i in range(iterations_):
    if start_key is not None:
        skip = 1
    result = db.view('all_docs',
                     startkey=start_key,
                     include_docs=True,
                     limit=chunk_size,
                     skip=skip)
    for row in result:
        doc = row["doc"]
```

Listing 5.38: Main migration loop

The workflow in the migration script follows the process of the Data Factory pipeline described in section 5.1. It transforms the data, saves the domain and stores metadata into `CosmosDB`. Optionally if no MID is present the script will try to resolve one. If no stats dictionary is present the update functionality of the Factserver will be invoked. On the other side, if a stats dictionary is included in the document, it is split on predefined keys into a stats and extended_stats schema to populate both files as described in listing 5.39.

```
if doc.get("stats") is not None:
    stats = {k: doc["stats"][k] for k in stat_keys}
    extended_stats = {k: doc["stats"][k] for k in extended_stats_keys}
    fs.set_stats(stats)
    fs.set_extended_stats(extended_stats)
else:
    update_script = Update()
    update_script.compare_and_save_stats(fs)
```

Listing 5.39: Conditional stats storage or computation

5.7. Refactoring

This section covers smaller changes to modules that already exist in the `CouchDB` framework.

5.7.1. Resolver

The `Resolver` class is used in the Factserver to retrieve lists of Factsets based on keys, for example all Factsets with a specific MID. These look-ups used to be queried from a dedicated second database.
In the `CouchDB` implementation the `Resolver` used a set of small, specialized views to achieve the lookup of `FactSets` from the database. While the `azure_facade` offers an

implementation for `query_view()` as described in section 5.3.4, implementing each view involves a lot of effort to keep the logic flexible to be able to respond to each possible parameter, even though this might not even be needed in the Factserver. Also, the complexity of the already large method grows with each addition. Since the views called by the `Resolver` are very simple, this opportunity was used to instead refactor the `Resolver` to call the new getters in the `azure_facade`, which are described in section 5.3.4, directly. For example, `Resolver.get_cache_by_source()` now calls `azure_facade.get_object_by_source()`. This only has the implication that the transformation towards the format which would be returned by the CouchDB and is already covered in `azure_facade.query_view()`, must be done by the `Resolver` itself. The main method used for this is shown in listing 5.40.

```
def _default_view_format(result):
    result_view_transformed = []
    for fs_json in result:
        transformed_json = {
            "id": fs_json["id"],
            "key": [
                fs_json["source"],
                fs_json["name"]
            ],
            "value": fs_json["mid"]
        }
        result_view_transformed.append(transformed_json)
    return result_view_transformed
```

Listing 5.40: Recreate CouchDB format

The `Resolver` class used to expose a `get_cache()` method which could retrieve Factsets by multiple optional parameters: `source`, `id`, `mid` and `factset`. However, only the `mid` parameter was used in the Factserver. For this reason, during the refactoring the method was streamlined and renamed into `get_cache_by_mid()`. While the Cache database is not a thing anymore, the naming is kept highlighting the similarity in usage.
The `resolve` method is designed to retrieve the MID either from the database or from the Google Knowledge Graph. This method used to take in an old Factset object as a parameter. However, as described in section 5.4, the new Factset is not just a wrapper for a JSON but instead expects the MID to be already present for the constructors. For this reason, the method was refactored to work on a dictionary instead of an object, which is enough for the purpose needed in this method which is storing a collection of values associated with the Factset. In practice, this results in multiple changes from attribute access using dot notation like `factset.name` to key-based access using square brackets as in `factset["name"]`.

5.7.2. FactManager

The `FactManager` class was originally intended to manage facts and interact with the database. It mainly holds multiple getter methods to retrieve Factsets from the database and therefore is ,now ,after the removal of the cache database described in section 4.3.3, similar in its intended purpose to the `Resolver` class described in the previous section.

For this reason, the refactoring approach also mirrors what was already described in the `Resolver` chapter: Simple view queries are replaced by `azure_facade` getters. However, while for the `Resolver` class query results had to be reformatted to fit the expected result from CouchDB, this is not necessary here because the returned result is a list of JSONs most of the time. The only formatting logic is shown in listing 5.41 and handles the `include_docs` parameter, which can be used to specify wether the full document or just a list of `ids` should be returned.

```
def _view_post_processing(result, include_docs):
    if include_docs:
        return result
    ids = []
    for row in result:
        ids.append(row['id'])
    return ids
```

Listing 5.41: Recreate include_docs parameter

Using the new appropriate constructor described in section 5.4, in `FactManager.get_by_id()` the line `Factset(self.server.get_object_by_id(id_))` was replaced by `Factset.from_id(id_)`.
Finally, the old `FactManager` would use a view called `basic/all` to retrieve every Factset for its `get_all` method. This has been replaced by a dynamically configured SQL query as shown in listing 5.42.

```
query = """SELECT """
if include_docs:
    columns = COSMOS_COLUMNS + " FROM c "
else:
    columns = "c.id FROM c "
offset_limit = "OFFSET @offset LIMIT @limit"
params = [
    {"name": "@limit", "value": limit},
    {"name": "@offset", "value": skip}
]
return azure_facade.execute_query(query+columns+offset_limit, params)
```

Listing 5.42: Query to get all Factsets based on parameter

Save and delete methods have been removed from the `FactManager` class and are now covered in `azure_facade` as described in section 5.3.4.

5.8. Testing

This section describes the different testing steps and their implementation implications in the Factserver.

5.8.1. Unit Tests

Unit tests are a common method for testing the correctness of small code components. For the Factserver the framework for unit testing chosen is `pytest`[5].
The change from `CouchDB` to Azure `CosmosDB` also required adaptation of the test environment. In the legacy system, the `_mock_couch` fixture simulated the `CouchDB` by mocking access to views and documents using `MagicMock`[6] objects. This included mocking `couchdb.Server`, `Database.view()` and methods like `__getitem__` and `__contains__`. This fixture was used then in the unit tests for the resolver module, since in the old implementation, the module held instances of the server and database itself. In contrast, the updated `_mock_azure_for_resolver` fixture mocks the `azure_facade` interface directly into the resolver module, for example by directly injecting the result returned by `factserver.resolver.azure_facade.get_object_by_source()` method as shown in listing 5.43, because the resolver module does not hold `CouchDB` instances anymore and communication is unified in the facade.

```
mock_get_object_return = [{'id': TEST_ID,
                           'mid': TEST_MID,
                           'source': TEST_SOURCE,
                           'name': TEST_NAME,
                           "type": TEST_TYPE}]
mocker.patch("factserver.resolver.azure_facade.get_object_by_source",
                    return_value=mock_get_object_return)
```

Listing 5.43: Mocking DB returns

Besides the unit tests for the resolver module, no changes to the already present mocking was necessary, because no other unit tests work with connections to the database.
Furthermore, a set of unit tests was written to cover a special case for Factset creation. Sometimes, then the IdaFix web extension ([1]) crawls multiple entities from the same webpage, the created dataset holds a graph value with a list of entity dictionaries. To ensure this logic was converted correctly without having to rely on the connection to IdaFix for testing the test case creates an example dictionary and checks that the transformation held the expected format. For this a preferred list of entities is mocked into the configuration of the Factserver as can be seen in listing 5.44.

```
@patch("factserver.factset.full_config", {"server": {"preflist": ["Person
    "]}})
def test_prefitem_match(self):
    fact_json = {
        "id": "97341824e191d5e332d5a29a1011f321",
        ...
        "data": {
            "@context": {"schema": "http://schema.org"},
            "@graph": [
                {"@id": "2", "@type": "schema:Thing", "name": "Box"},
                {"@id": "1", "@type": "schema:Person", "name": "Alice"}
            ]
```

[5] https://docs.pytest.org/en/stable/
[6] https://docs.python.org/3/library/unittest.mock.html

```
12          }
13      }
14      factset = Factset.from_json(fact_json)
15      self.assertIsNotNone(factset._data)
16      self.assertEqual(factset._data["@id"], "1")
17      self.assertEqual(factset._data["@type"], "schema:Person")
18      self.assertIn("@context", factset._data)
```

Listing 5.44: Testing graph transformation logic

Finally, unit tests were written to cover the `query_view` logic described in section 5.3.4. They are intended to check if the returned formats match the expectations under the parameters which could be provided to CouchDB's method. In listing 5.45 mock data consisting of a list of `sources` is inserted into the database. The unit test checks if the result if formatted and filtered correctly when the `key` parameter is provided and matches only one of the inserted dictionaries.

```
1 @patch.object(AzureFacade, "_container")
2 def test_query_view_with_key(mock_container):
3     mock_data = [{"source": "https://example.com/page1"}, {"source": "
      https://another.com/page2"}]
4     mock_container.query_items.return_value = mock_data
5
6     result = af.query_view("basic/domain", key="example.com")
7
8     assert len(result) == 1
9     assert result[0]["key"] == "example.com"
```

Listing 5.45: Testing Query View filter

5.8.2. Schemathesis

`Schemathesis`[7] is an automated testing framework for RESTful APIs. By analyzing the OpenAPI schema of the Factserver, `Schemathesis` generates a wide variety of inputs such as edge cases and malformed data and sends them to the endpoints, while not relying on fixed or predefined inputs.
To comply with the `Schemathesis` requirement to reply to HTTP requests to endpoints with not specified methods with a 405 status code, something that is not natively supported by Azure Functions, the Factserver already implemented a `is_request_valid()` method. To further enhance the ability of endpoints to automatically handle malformed requests, this method has been extended to automatically check the request body for validity on POST requests as is described in listing 5.46. For example, `Schemathesis` tries to break endpoints by inserting byte-strings that the API logic can not handle.

```
1 if req.method == "POST":
2     if req.headers.get("Content-Type") == "application/json":
3         try:
4             req_body = req.get_json()
5             if req_body is None or not isinstance(req_body, dict):
```

[7] `https://schemathesis.io/`

```
6                 raise ValueError("Body is not a valid JSON object")
7             data = req_body.get("data", None)
8             if not data:
9                 return True, "OK"
10            if not isinstance(data.get("name", ""), str) or not
    isinstance(data.get("@type", ""), str):
11                raise ValueError("@type or name is not a valid string")
```

Listing 5.46: Validation of POST bodies

5.8.3. Integration Testing

While `Schemathesis` validates the syntax and schema of the endpoints, the framework is unable to verify semantic correctness in writes to database, Parquet file creation, statistic calculation etc.
To ensure all endpoints work as expected with the new storage framework, a minimal dataset was created with two Factsets providing information for the same MID but different sources so that the creation of comparison statistics will not be skipped. While the data dictionary for this test can be kept brief, to check all the functionality at least one field should have the same name but a different value between both Factsets. An example dataset is provided in the appendix A.3
First, the dataset is copied into a .txt file and uploaded to the staging folder in the raw bucket to trigger the data factory pipeline. The Azure web interface can be used to verify that the pipeline calls to the transformation endpoints described in section 5.5 worked by looking at the metadata in the `CosmosDB` and inspecting the domain.txt file and the Parquet files for data, stats, extended_stats and fact_stats. Using the `id` and `mid` from the `CosmosDB` and the `type`, `name` and `source` the following endpoints can now be manually checked without further setup for expected results for example using Postman or `curl`.

- GET /facts
- GET /facts/fact_id
- POST /resolver
- GET /cache
- GET /cache/fact_id
- GET /domains
- GET /stats
- GET /stats/fact_id
- GET /stats/numfacts

The POST /compare and POST /compare/add endpoints can now be tested. To ensure the comparisons in the endpoint are triggered the data dictionary for the body needs to include the same `name` and `@type`, otherwise the FactServer will recognize that there are no `FactSet`s in the database for comparison.
The POST /historical can be triggered with a `date` at most 90 days after the date in the dataset and the domain from the dataset. For our example `2024-06-01` and `adria.ign.com` would trigger the calculation. Once this calculation is done the GET /historical endpoint can be tested.
In the future, an automatable test suite for integration testing should be written. However, this is out of the scope of this work for two reasons: Out of the 9 API controllers of the Factserver 6 create connections to the database and would therefore need to be included in integration testing, increasing the effort beyond what is feasible in this work. 2. For a clean integration testing framework, it needs to be ensured that the tests do not interfere with production data. Therefore, additional environments need to be created that can be used during testing in development as well as when the test suit is picked up by the GitLab runner for continuous development. The possibly automated setup of this environment is also beyond the scope of this work.

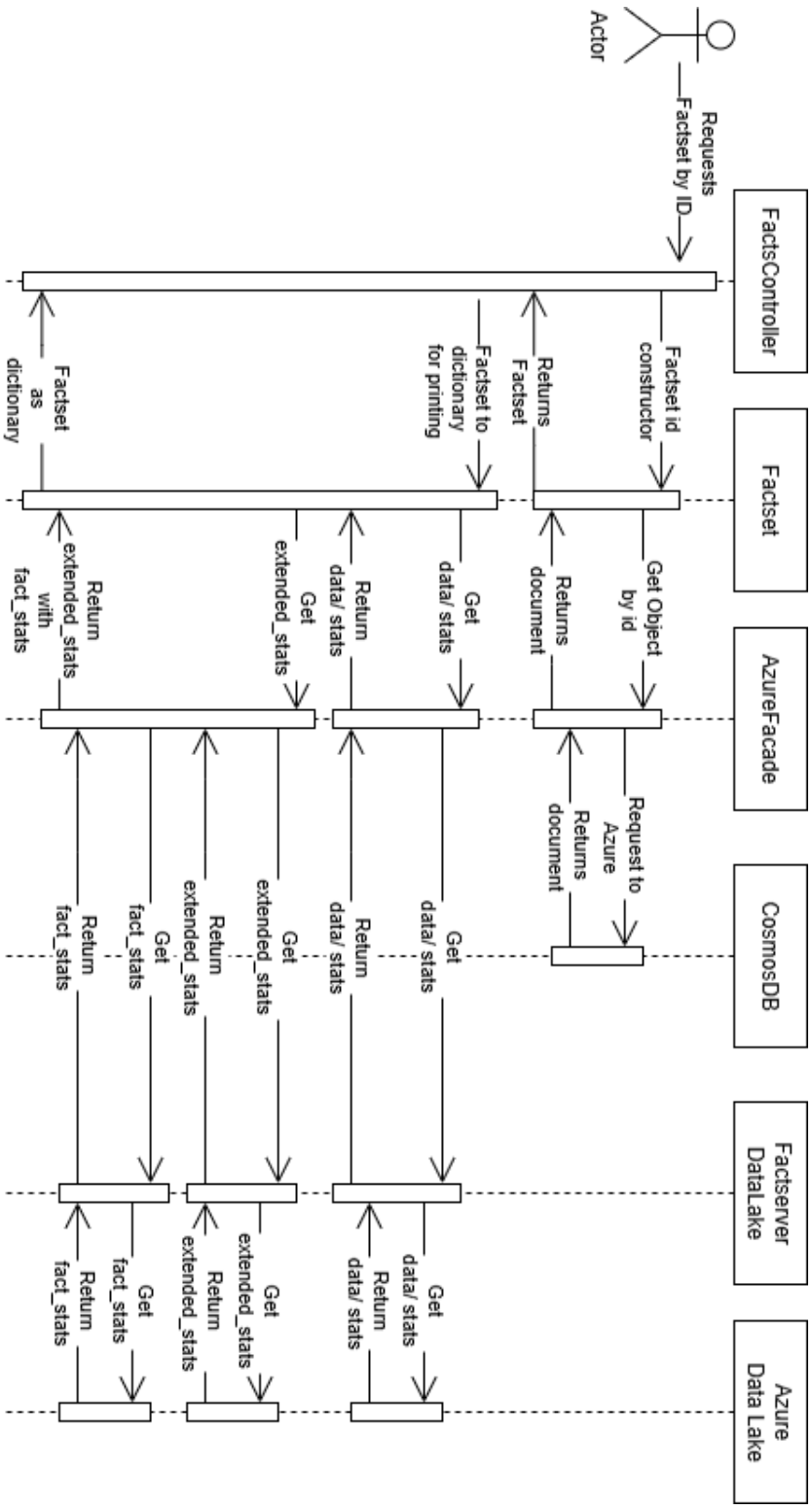

Figure 5.3.: Process view of full Factset retrieval by ID

6. Evaluation

While the effectiveness and usability of the implemented storage framework will be determined by the adoption in the Factcheck project over time, this section discusses those results of the work that can be evaluated immediately.

6.1. Cost and runtime evaluation for ADF pipeline

One of the main goals of this project was to investigate potential cost-saving when using ADF for the data transformation pipeline by using a SHIR instead of the default IR to reduce the amount of computation done in the cloud. The exact costs are listed in section 4.3.1.
An experiment was conducted to evaluate this. The SHIR was installed and configured on a personal laptop and used to execute the pipeline with a dataset size of 661 KB. The device specifications used are an AMD Ryzen 7 5700U prozessor with Radeon Graphics (1.80 GHz) and 16 GB of RAM. The pipeline was triggered at 2:54:24 pm and completed at 3:46 am, resulting in a runtime of approximately 13 hours. The cost reported in the Azure billing overview was €3.60.
For comparison, a second run was executed using the Azure IR. Because in this environment pipelines are not allowed to call `localhost` from web activities, the Factserver was deployed into the Azure environment itself, instead of as a docker container like how it is set up in production and for the rest of testing. Due to concerns about high costs and long runtime, which arose after the tests with the SHIR, the dataset was reduced to 66 KB, a tenth of the original size. This run completed in approximately 21 minutes but resulted in a significantly higher cost of 25.04€. While this setup uses Azure Functions, none of the costs are registered for this resource, meaning 100% of the 25.04€ are attributed to ADF. Under the assumption that the costs grow linearly, cost normalization highlights a strong price difference between the two environments.

SHIR:

$$\frac{3.60}{661\,\text{KB}} \approx 0.00544\,€/\text{KB}$$

Azure IR:

$$\frac{25.04}{66\,\text{KB}} \approx 0.3794\,€/\text{KB}$$

This shows that per kilobyte the Azure IR is roughly 70 times more expensive in this experiment.

Table 6.1.: Average activity duration over 10 runs

Activity name	Duration in sec IR	Duration in sec SHIR
Calculate Stats	11.2	25.04
Check MID	9.6	29.65
Save Domain	8.5	30.19
Sink Data to DL	6.6	38.96
Sink to CosmosDB	6.4	32.22
query Google KnowledgeGraph	6.0	20.02

The extended runtime for the SHIR pipeline is likely influenced by local hardware and Wi-Fi connection speed. While even in an optimized physical environment long runtime is to be expected, it could be acceptable if enough work can be managed overnight.
These exact numbers should be taken with a grain of salt as the sample size is limited and the results could be heavily influenced by the available hardware, but still the results clearly indicate the cost-saving potential of using a setup with the SHIR. For production use, further testing on dedicated hardware is recommended.
Finally, a comparison of the average execution times of the single pipeline activities for the first 10 runs is shown in table 6.1. When comparing the activity runtime between the SHIR and IR it can be seen that tasks involving larger data transfers, such as 'Sink Data to DL' or 'Sink to CosmosDB' take relatively longer in the SHIR. This performance difference is expected, as data must be transferred between the local machine and Azure services, introducing network overhead. In contrast, within the Azure environment all operations occur inside the cloud infrastructure, allowing for faster data movement. This carries implications for potential future implementations and extensions of ADF pipelines, as the opportunity cost regarding speed for using SHIR grows as more data movements are included in the pipeline.

6.2. Discussion on ADF development and tradeoffs

ADF provides some advantages in this architecture. Built-in features like retry policies and monitoring dashboards help with resilience and observability. The ability to move parts of the workload away from the Factserver could also reduce the risk of a computational bottleneck, which might be valuable in large-scale deployments. Additionally, the modular structure of ADF pipelines encourages reusability and clear orchestration.
On the other hand, the integration of ADF into the architecture introduces a noticeable level of additional complexity. While it enables a clear separation of concerns between the transformation steps and the Factserver, this benefit comes at the cost of increased infrastructure management requirements. As discussed in section 5.1.1, limitations specific to the self-hosted integration runtime, such as the inability to execute 'data flow' activities, reduce the benefit of ADF in this setting. The cost of data movements remains high even with a SHIR as the pipeline involves repeated uploads and downloads to the different storage components. Developing for Azure Data Factory can be challenging and, at

times, frustrating due to the limited debugging capabilities, inconsistent behavior between activities and across IR, a lack of flexibility in the provided and custom logic and little available documentation, examples and discussions. In the future, it should be evaluated if transformations could be done in alternative locations while eliminating intermediate steps, for example by having the Factserver, or a potentially new setup such as Databricks, listen for and consume Event Grid topics.

7. Future Work

This chapter suggests research and new additions based on the presented work.

7.1. Alternatives to ADF

As discussed in section 6.2, future work should investigate possible replacements for ADF to reduce costs and increase processing efficiency. One opportunity is to use the existing Factserver infrastructure to handle the orchestration itself. The implementation already includes an `upload` script which is very similar to the migration script explained in section 5.6 but omits the checkpoint logic and is targeted at developers aiming to quickly create test data in the azure environment. However, combining this logic with the Azure Event Grid API for Python described in [31], a potential implementation of the Factserver consuming events itself without the need for ADF could look like listing 7.1. The path that triggered the `BlobCreated` event is parsed from the event subject. After this the file just must be downloaded and parsed.

```
@app.function_name(name="eventGridTrigger")
@app.event_grid_trigger(arg_name="event")
def transformRawFactset(event: func.EventGridEvent):
    path = event.subject.split("/blobs/", 1)[1]
    buffer = BytesIO()
    raw_blob.download_file(path, buffer)
    lines = buffer.decode('utf-8').splitlines()
    raw_factsets = [json.loads(line) for line in lines]
    upload(raw_factsets)
```

Listing 7.1: Factserver consuming Azure events draft

7.2. Optimize Bulk Operations

In the current setup, every time a new Factset is processed, the system loads the full Parquet file from the DL, appends the new data and uploads the whole file again. While this ensures the data to be always up-to-date, it creates a lot of overhead and could be inefficient for certain use cases based on their varying requirements. Because operations for potential optimization need to be tailored to specific project needs, they do not fit in the scope of this work. In the future, the Factservers `datalake` module may provide the option to delay uploads and collect content, for example by configuring the amount of write operations to be performed beforehand. Listing 7.2 displays a potential draft for these changes, including an internal counter and internal storage of intermediate appended

data. A prerequisite for this example would be to have the writing process to order the raw data, grouping them by name and type to ensure bulks of data can be written into the same partition.

```
    def append_df_to_dl_parquet(self, df, path: str, bulk_upload=None:
    int) -> None:
        buffer_new = BytesIO()
        buffer_old = BytesIO()
        if bulk_upload is not None:
            buffer_old = self.cached_bulk_buffer
        try:
            # old download  and combination logic
            ...
        if bulk_upload is not None:
            self.bulk_upload_counter += 1
            if self.bulk_upload_counter == bulk_upload:
                self.upload_buffer(buffer_new, path)
                self.bulk_upload_counter = 0
            return None
        self.upload_buffer(buffer_new, path)
```

Listing 7.2: Bulk upload draft

The reduced write frequency could speed up operations and make them more cost-effective. Another approach would be to allow time-based uploads or return to the Azure Queue setup already running in the Factserver. Any such improvements need to be optional as to not interrupt existing projects expecting immediate consistency.

7.3. Refactoring Resolver into Factmanager

Another idea for future work lies in the combination of the `Resolver` and `Factmanager` classes. Both currently provide overlapping functionality with methods created to retrieve Factset documents based on attributes or combinations such as `id`, `mid`, `name` or `source` such as `get_cache_by_mid` for `Resolver` and `get_json_by_mid` for the `FactManager`. The `Resolver` class used to access a separate database named "cache" in the `CouchDB` setup, while the `FactManager` connected to the main Factset database. Without a clear case for separation they have been refactored to retrieve information from the same, main database in `CosmosDB` for the Azure storage framework. Refactoring the `Resolver` into the `Factmanager`, or creating a combination of both, would reduce redundancy, simplify testing and clarify the systems architecture. It may also make it easier to extend the system in the future.

7.4. Remove include docs

Optimization potential for future work is in the complete removal of the `include_docs` parameter scattered in the Factserver and left over from `CouchDB` design. Originally, this parameter was used to switch between full document retrieval or just lists of document IDs from `CouchDB` views. However, this functionality is not natively supported in `CosmosDB`.

For many `query_view()` calls in the Factserver the decision for the `include_docs` value is provided by the end user of the API rather than based on internal use cases of the Factserver. In the context of Azure, simulating this behavior adds complexity. Additionally, dynamically constructing SQL queries to either return full documents or just IDs worsens the `AzureFacade`s separation of concern. Wether a method operates on entire documents or just IDs reflects different use cases and should not be decided at runtime.

> Functions should do one thing. - Robert Martin in [29, Ch. 3, p. 65]

Splitting these responsibilities leads to cleaner code, better readability and clearer potential for optimization.
One concern which led to the inclusion of the `include_docs` parameter in methods used to be performance then retrieving large amounts of full documents. This is first not an issue anymore with the Factserver only retrieving the metadata from `CouchDB` and secondly, with the getters described in section 5.3.4, the behavior of only retrieving IDs is only simulated after the query execution, eliminating the opportunity for performance improvements completely.

7.5. Data Lifecycle Management

The current implementation of the data storage framework does not include any mechanisms for data lifecycle management. Once data is uploaded it remains in the DL permanently, unless it is manually deleted. While this approach works during development and in the short-term, it could become a problem in the production environment, for example due to uploads from real-time web crawlers or production level, manual data upload. Therefore, eventually it will become necessary to define and implement rules for how long different types of data should be stored.
For example, raw data could be automatically deleted after a certain time, while enriched Factsets or statistics should be stored longer. Azure DL includes options for tiered storage and lifecycle policies, which could be explored to reduce storage costs and avoid clutter. For example, blobs that haven't been accessed for the last 30 days can be automatically moved to a lower cost storage tier that in return needs more time to retrieve files on request [34]. Another approach would be to run custom cleanup to remove data associated with older dates out of Parquet files while keeping the rest of the file in place. This could be automated using schedulers like Apache Airflow or simple `cron` jobs.
The current partitioning strategy by `/type/name` might prove not fine-grained enough for some use cases. To remedy this, one step in data lifecycle management might be to repartition data by including additional fields or dates.
Introducing these mechanisms would make the framework more scalable by reducing retrieval time then the Factserver must perform lookups between large Parquet files. The above measures for removing data volume would also be an important step to keep growing cloud costs in check. Another benefit is that downstream applications like dashboards or web extensions will have to perform less filtering operations to display the most up-to-date insights.

7.6. Integration Testing

As mentioned in section 5.8.3, integration testing was not included in the scope of this thesis due to the size of the Factserver's API and the risk of interfering with production data. In the future, dedicated integration tests should be developed to ensure that the interaction between the components, such as the API endpoints, `CosmosDB` and the DL works reliably. This would include setting up a separate environment with duplicated infrastructure. Automating this setup, for example with Terraform or ARM templates is necessary to ensure reproducibility between projects and within CI pipelines in GitLab. Terraform can be invoked as part of the integration testing stage to provision a testing environment on demand for a pipeline (see for example [53]). The pipeline could have a job that uses Terraform to create a new set of Azure resources, the Factserver's integration tests would then execute against this environment and afterwards Terraform destroys the resources to tear it down again. This would ensure minimal cost and no interference between multiple test runs.

8. Conclusion

This thesis explored the question how the data storage for the FactCheck framework, a system for automatic comparison of semi-structured data, could benefit from a migration to Azure's cloud infrastructure. Drawbacks of the existing CouchDB implementation were slow data retrieval as well as limited extendibility in development. The theoretical background for the work was examined in chapter 2. 39 papers were examined using the method PRISMA. One key focus of state of the art research on data storage frameworks was identified, namely the LH, which combines the transactional strenghts of the DW with the cheap storage of the DL.
Chapter 3 considered the requirements for the data storage framework. It also analyses the `CouchDB` design to lay the foundation for the changes developed.
Chapter 4 answers the first research question 1 by introducing the design of the new storage framework and discussing the architectural decisions around the components. The architecture consists of an Azure DL and Azure `CosmosDB` where data cleaned in an ADF pipeline is stored. The schema of the central Factset entity was changed to match the requirements of this new system. Additionally, the key changes to enable the Factserver to communicate with this new system, such as the `AzureFacade` and `DataLake` class were designed. One mayor redesign is the separation of the `data` and `stats` dictionaries from the document storage into the DL to reduce storage costs.
Chapter 5 covered the implementation of the transformation pipeline and the necessary changes for the Factserver to communicate with the Azure infrastructure, like new endpoints, file up- and download and database access. It acted as the Proof of Concept requested in the second research question 2. One key achievement was to preserve all the functionality of the Factserver that relied on the database. In most cases the same method signatures can be used without changes. However, some modules had to be refactored and a few access pattern changed.
The discussion of the implementation using ADF was done in Chapter 6. While the comparison between Azure and self-hosted environments showed an improvement in cost per KB transformed, the result might still be to expensive for a production system under the given requirements. The general fit of ADF for the use case was questioned.
Continuing with this discussion, Chapter 7 presented possible future work, starting with a suggestion for an alternative implementation for the transformation pipeline entirely in the Factserver itself, avoiding expensive data transportation work. Additionally, options to reduce complexity in the Factserver and optimize transactions were introduced.
The answer targeting the potential constraints raised in the third research question 3 was split between Chatpers 3, 4 and the evaluation in 5. Some potential remedies were included in Chapter 6. The answer to 4 was provided in Chapter 4.4
This work contributes towards replacing `CouchDB` with a reliable data storage framework

in Azure that facilitates fast querying. During the transformation all the functionality of the Factserver was kept intact. Wether users and developers are able and willing to adapt this new storage framework will be tested over the next release cycles.

Bibliography

[1] Mara Sophie Aichinger. Towards a cloud-native framework for resolving issues in (semi-)structured data with human feedback in the context of FactCheck++, 2023.

[2] Soukaina Ait Errami, Hicham Hajji, Kenza Ait El Kadi, and Hassan Badir. Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse. *Journal of parallel and distributed computing*, 176:70–79, 2023.

[3] Sarah Azzabi, Zakiya Alfughi, and Abdelkader Ouda. Data Lakes: A Survey of Concepts and Architectures. *Computers*, 13(7), 2024.

[4] Daniel Bauer, Florian Froese, Luis Garcés-Erice, Chris Giblin, Abdel Labbi, Zoltán A. Nagy, Niels Pardon, Sean Rooney, Peter Urbanetz, Pascal Vetsch, and Andreas Wespi. Building and Operating a Large-Scale Enterprise Data Analytics Platform. *Big Data Research*, 23, 2021.

[5] Edmon Begoli, Ian Goethert, and Kathryn Knight. A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks. *Proceedings - 2021 IEEE International Conference on Big Data, Big Data 2021*, page 4643 – 4651, 2021. All Open Access, Green Open Access.

[6] Ivo Blohm, Felix Wortmann, Christine Legner, and Felix Köbler. Data products, data mesh, and data fabric. *Business information systems engineering*, 66(5):643–652, 2024.

[7] I. Bojicic, Z. Marjanovic, N. Turajlic, M. Petrovic, M. Vuckovic, and V. Jovanovic. Domain/mapping model: A novel data warehouse data mode. *International Journal of Computers Communications Control*, 12(2):166–182, 2017.

[8] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian Sun, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. Shared Foundations: Modernizing Meta's Data Lakehouse. *13th Annual Conference on Innovative Data Systems Research, CIDR 2023*, 2023.

[9] E. F. Codd. A relational model of data for large shared data banks, 1970.

[10] Z. Dehghani. How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh. https://martinfowler.com/articles/data-monolith-to-mesh.html, 2019. Accessed: 2024-11-24.

[11] Anton Dolhopolov, Arnaud Castelltort, and Anne Laurent. Implementing Federated Governance in Data Mesh Architecture. *Future internet*, 16(4):115, 2024.

[12] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Leveraging the Data Lake: Current State and Challenges. In Carlos Ordonez, Il-Yeol Song, Gabriele Anderst-Kotsis, A Min Tjoa, and Ismail Khalil, editors, *Big Data Analytics and Knowledge Discovery*, pages 179–188, Cham, 2019. Springer International Publishing.

[13] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Implementation Patterns for Zone Architectures in Enterprise-Grade Data Lakes. In *Advanced Information Systems Engineering*, volume 14663 of *Lecture Notes in Computer Science*, pages 267–283. Springer, Switzerland, 2024.

[14] Roelien Goede. Data vault modelling as alternative to dimensional modelling to embrace complexity in data driven decision systems: a critical systems perspective. In *2022 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–5, 2022.

[15] Abel Goedegebuure, Indika Kumara, Stefan Driessen, Willem-Jan Van Den Heuvel, Geert Monsieur, Damian Andrew Tamburri, and Dario Di Nucci. Data Mesh: A Systematic Gray Literature Review. *ACM Comput. Surv.*, 57(1), October 2024.

[16] Jay Gordon. Azure Cosmos DB design patterns – Part 1: Attribute array. https://devblogs.microsoft.com/cosmosdb/azure-cosmos-db-design-patterns-part-1-attribute-array/, 2023. Accessed: 2025-04-27.

[17] Ahmed A. Harby and Farhana Zulkernine. In *2022 IEEE International Conference on Big Data (Big Data)*.

[18] Ahmed A. Harby and Farhana Zulkernine. From Data Warehouse to Lakehouse: A Comparative Review. *Proceedings - 2022 IEEE International Conference on Big Data, Big Data 2022*, page 389 – 395, 2022.

[19] Ahmed A. Harby and Farhana Zulkernine. Data Lakehouse: A survey and experimental study. *Information Systems*, 127, 2025.

[20] Gilbert Held. *A practical guide to content delivery networks*. CRC Press, Boca Raton, second edition. edition, 2011.

[21] Dieter Hoffmann. *Data Warehouse im Rahmen der Business Intelligence : Konzeption eines Vorgehensmodells*. Diplomica Verlag,, Hamburg :, 1st ed. edition, 2010.

[22] William H Inmon. *Building the data warehouse*. Wiley computer publishing. Wiley, New York, NY [u.a.], 2. ed. edition, 1996.

[23] Ralph Kimball. *The Kimball Group Reader : Relentlessly Practical Tools for Data Warehousing and Business Intelligence*. John Wiley Sons, Incorporated,, Hoboken :, 1st ed. edition, 2010.

[24] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015*, pages 2785–2792, 2015.

[25] Mark Kromer. New Data Flow Connector: SQL Server as Source and Sink. https://techcommunity.microsoft.com/blog/azuredatafactoryblog/new-data-flow-connector-sql-server-as-source-and-sink/2406213, 2021. Accessed: 2025-10-12.

[26] N. G. Kuftinova, O. I. Maksimychev, A. V. Ostroukh, A. V. Volosova, and E. N. Matukhina. Data Fabric as an Effective Method of Data Management in Traffic and Road Systems. In *2022 Systems of Signals Generating and Processing in the Field of on Board Communications*, pages 1–4, 2022.

[27] Dan Linstedt and Michael Olschimke. *Building a Scalable Data Warehouse with Data Vault 2.0.* Elsevier Inc., 2015.

[28] Inês Araújo Machado, Carlos Costa, and Maribel Yasmina Santos. Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures. In *Procedia Computer Science*, volume 196, pages 263–271. Elsevier B.V, 2022.

[29] Robert C Martin. *Clean Code : Refactoring, Patterns, Testen und Techniken für sauberen Code ; [Kommentare, Formatierung, Strukturierung, Fehler-Handling und Unit-Tests, zahlreiche Fallstudien, Best Practices, Heuristiken und Code Smells].* mitp, Heidelberg München, Landsberg, Frechen, Hamburg, deutsche ausgabe edition, 2009.

[30] Nathan Marz. How to beat the CAP theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html, 2011. Accessed: 2024-11-24.

[31] Microsoft. Azure Event Grid trigger for Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-grid-trigger?tabs=python-v2 Accessed: 2025-07-21.

[32] Microsoft. Caching with Azure Front Door. https://learn.microsoft.com/en-us/azure/frontdoor/front-door-caching?pivots=front-door-standard-premium, 2023. Accessed: 2025-07-22.

[33] Microsoft. Including and excluding property paths. https://learn.microsoft.com/en-us/azure/cosmos-db/index-policy, 2024. Accessed: 2025-04-27.

[34] Microsoft. Configure a lifecycle management policy. https://learn.microsoft.com/en-us/azure/storage/blobs/lifecycle-management-policy-configure?tabs=azure-portal, 2025. Accessed: 2025-07-23.

[35] Microsoft. Configure caching on Azure Front Door. https://learn.microsoft.com/en-us/azure/frontdoor/how-to-configure-caching, 2025. Accessed: 2025-07-22.

[36] Microsoft. Create and configure a self-hosted integration runtime. https://learn.microsoft.com/en-us/azure/data-factory/create-self-hosted-integration-runtime?tabs=data-factory, 2025. Accessed: 2025-05-02.

[37] Microsoft. Data Pipeline pricing. https://azure.microsoft.com/en-us/pricing/details/data-factory/data-pipeline/, 2025. Accessed: 2025-05-02.

[38] Microsoft. ForEach activity in Azure Data Factory and Azure Synapse Analytics. https://learn.microsoft.com/en-us/azure/data-factory/control-flow-for-each-activity, 2025. Accessed: 2025-10-12.

[39] Natalia Miloslavskaya and Alexander Tolstoy. Big Data, Fast Data and Data Lake Concepts. In *Procedia Computer Science*, volume 88, pages 300–305. Elsevier B.V, 2016.

[40] Athira Nambiar and Divyansh Mundra. An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management. *Big data and cognitive computing*, 6(4):132, 2022.

[41] Abhishek Narain. Web activity response timeout improvement. https://techcommunity.microsoft.com/blog/azuredatafactoryblog/web-activity-response-timeout-improvement/3260307, 2022. Accessed: 2025-10-12.

[42] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities. *Proceedings of the VLDB Endowment*, 12(12):1986–1989, 2019.

[43] Institute of Electrical and issuing body. Electronics Engineers, author. An Overview of Current Data Lake Architecture Models. *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO) /*, pages 1082–1087, 2022.

[44] Dražen Oreščanin and Tomislav Hlupić. Data Lakehouse - a Novel Step in Analytics Architecture. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1242–1246, 2021.

[45] Matthew J. Page, Joanne E. McKenzie, Patrick M. Bossuyt, Isabelle Boutron, Tammy C. Hoffmann, Cynthia D. Mulrow, Larissa Shamseer, Jennifer M. Tetzlaff, Elie A. Akl, Sue E. Brennan, Roger Chou, Julie Glanville, Jeremy M. Grimshaw, Asbjørn Hróbjartsson, Manoj M. Lalu, Tianjing Li, Elizabeth W. Loder, Evan Mayo-Wilson, Steve McDonald, Luke A. McGuinness, Lesley A. Stewart, James Thomas, Andrea C. Tricco, Vivian A. Welch, Penny Whiting, and David Moher. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. *Systematic reviews*, 10(1):89–89, 2021.

[46] Torsten Priebe, Sebastian Neumaier, and Stefan Markus. Finding Your Way Through the Jungle of Big Data Architectures. *2021 IEEE International Conference on Big Data (Big Data)*, pages 5994–5996, 2021.

[47] Sean Rooney, Daniel Bauer, Luis Garces-Erice, Peter Urbanetz, Florian Froese, and Sasa Tomic. Experiences with managing data ingestion into a corporate datalake. In *Proceedings - 2019 IEEE 5th International Conference on Collaboration and Internet Computing, CIC 2019*, page 101 – 109. Institute of Electrical and Electronics Engineers Inc., 2019.

[48] Code Sample. Azure Cosmos DB design pattern: Attribute array. https://learn.microsoft.com/en-us/samples/azure-samples/cosmos-db-design-patterns/attribute-array/, 2024. Accessed: 2025-04-27.

[49] Pegdwendé Sawadogo and Jérôme Darmont. On data lake architectures and metadata management. *Journal of intelligent information systems*, 56(1):97–120, 2021.

[50] Jan Schneider, Christoph Gröger, Arnold Lutsch, Holger Schwarz, and Bernhard Mitschang. The Lakehouse: State of the Art on Concepts and Technologies. *SN computer science*, 5(5):449, 2024.

[51] Jan Schneider, Christoph Gröger, Arnold Lutsch, Holger Schwarz, and Bernhard Mitschang. The Lakehouse: State of the Art on Concepts and Technologies. *SN Computer Science*, 5(5), 2024. All Open Access, Hybrid Gold Open Access.

[52] Roman Szabo. FactCheck++ reference architecture based on Azure Cloud Services, 2022.

[53] Terraform. Automate Terraform with GitHub Actions. https://developer.hashicorp.com/terraform/tutorials/automation/github-actions, 2025. Accessed: 2025-10-12.

[54] Daniel Tovarnak, Matus Racek, and Petr Velan. Cloud Native Data Platform for Network Telemetry and Analytics. In Chemouil P., Ulema M., Clayman S., Sayit M., Cetinkaya C., and Secci S., editors, *Proceedings of the 2021 17th International Conference on Network and Service Management: Smart Management for Future Networks and Services, CNSM 2021*, page 394 – 396. Institute of Electrical and Electronics Engineers Inc., 2021.

[55] Christopher Vox, David Broneske, Jan Piewek, Janusz Feigel, and Gunter Saake. Investigating Lakehouse-Backbones for Vehicle Sensor Data. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14146 LNCS:243 – 258, 2023.

[56] Dan Woods. Big data requires a big architecture. https://www.forbes.com/sites/ciocentral/2011/07/21/big-data-requires-a-big-new-architecture/, 2011. Accessed: 2024-11-24.

[57] Qi Xiao, Wenkui Zheng, Chenyu Mao, Wei Hou, Hao Lan, Daojun Han, Yang Duan, Peng Ren, and Ming Sheng. MHDML: Construction of a Medical Lakehouse for Multi-source Heterogeneous Data. *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 13705 LNCS:127 – 135, 2022.

[58] Behrouz Zolfaghari, Gautam Srivastava, Swapnoneel Roy, Hamid R. Nemati, Fatemeh Afghah, Takeshi Koshiba, Abolfazl Razi, Khodakhast Bibak, Pinaki Mitra, and Brijesh Kumar Rai. Content Delivery Networks: State of the Art, Trends, and Future Roadmap. *ACM computing surveys*, 53(2):1–34, 2021.

[59] Katerina Černjeka, Danijela Jakšić, and Vladan Jovanovic. NoSQL document store translation to data vault based EDW. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1197–1202, 2018.

Acronyms

ACID atomicity, consistency, isolation, durability. 9, 12, 14

ADF Azure Data Factory. ix, 28, 29, 36, 67, 68, 71, 75

AFD Azure Front Door. 35

AI Artificial Intelligence. 11, 13, 20

ARM Azure Resource Manager. 36

CDN Content Delivery Network. 2, 3, 5, 19, 20, 35

DL Data Lake. 6, 10–14, 16–18, 20, 28, 31–33, 36, 40, 42–45, 47, 49, 51, 54, 55, 57, 59, 71, 73–75

DW Data Warehouse. 6, 8–17, 20, 28, 75

ER Entity-Relationship. 7

ETL extract-transform-load. 10, 12

HDFS Hadoop Distributed File System. 16

IR Integration Runtime. xi, 29, 30, 67–69, 87

LH Data Lakehouse. 6, 11–16, 20, 28, 75

ML Machine Learning. 10, 11, 13, 15, 18, 19

OO object-oriented. 7

RA reference architecture. 36

RDBMS Relational Database Management System. 7

SHIR Self-Hosted Integration Runtime. xi, 29, 30, 39, 67, 68, 88

TTL time-to-life. 19

A. Appendix

A.1. Repositories

Factserver Fork `https://git01lab.cs.univie.ac.at/arthurr96/fact-server-fork-12035966`

Infrastructure templates `https://git01lab.cs.univie.ac.at/thesis/2024ws/12035966_riess-arthur_24w`

Raw crawler factsets `https://git01lab.cs.univie.ac.at/factcheck/data/facts`

A.2. Example domain.txt Content

wiki.univie.ac.at
adria.ign.com
de.ign.com
fakewebsite.com
fakewebsite2.com
www.example.org
www.hulu.com

A.3. Example Queries

In this section example formats for multiple results inside of the FactServer are presented.

```
SELECT * FROM c WHERE c.type = @book OFFSET 2 LIMIT 4
```

Listing A.1: Example SQL query dyamically constructed inside the FactServer

```
{
"types" : [Person, Book, Movie],
"count" : 3
}
```

Listing A.2: Example CouchDB Output dyamically constructed inside the FactServer

A.4. Example Dataset for Integration Testing

This section shows an example dataset used for Integration Testing. It holds two JSON which will be split into two Factsets during upload. Two enable a wide range of functionality to be tested, they cover the same entity, but vary slightly in the information included.

```
{
"date": "2024-05-13T13:51:20.706823",
"userid": 999,
"mid": null,
"stats": null,
"source": "https://adria.ign.com/assassins-creed-film/11847/review
   /assassins-creed-recenzija",
"type": "https://schema.org/Movie",
"data": {
    "@context": "https://schema.org",
    "@type": "Movie",
    "name": "Assassin's Creed: The Movie",
    "description": "example description"
  }
}
{
"date": "2024-05-13T13:51:20.706823",
"userid": 999,
"mid": null,
"stats": null,
"source": "https://examplewebpage.com/assassins-creed-film",
"type": "https://schema.org/Movie",
"data": {
    "@context": "https://schema.org",
    "@type": "Movie",
    "name": "Assassin's Creed: The Movie",
    "description": "different description"
  }
}
```

Listing A.3: Example Dataset

A.5. ADF runtime raw data

Activity name	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Calculate Stats	13.0	3.0	4.0	13.0	14.0	13.0	13.0	13.0	13.0	13.0
Check MID	8.0	17.0	8.0	7.0	7.0	18.0	8.0	7.0	8.0	8.0
Save Domain	3.0	14.0	3.0	3.0	5.0	14.0	13.0	14.0	3.0	13.0
Sink Data to DL	4.0	3.0	3.0	4.0	4.0	3.0	14.0	3.0	14.0	14.0
Sink to CosmosDB	13.0	4.0	13.0	3.0	4.0	3.0	3.0	14.0	4.0	3.0
query Google KnowledgeGraph	4.0	14.0	5.0	4.0	4.0	14.0	4.0	3.0	4.0	4.0

Table A.1.: All Durations By Run in seconds(Azure IR)

Activity name	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Calculate Stats	9.0	110.0	8.0	11.0	19.0	164.0	13.0	17.0	13.0	9.0
Check MID	21.0	21.0	19.0	130.0	21.0	29.0	15.0	23.0	23.0	24.0
Save Domain	15.0	24.0	119.0	17.0	21.0	13.0	18.0	15.0	20.0	116.0
Sink Data to DL	12.0	15.0	11.0	12.0	14.0	118.0	8.0	13.0	13.0	32.0
Sink to CosmosDB	7.0	8.0	11.0	11.0	8.0	80.0	110.0	7.0	9.0	20.0
query Google KnowledgeGraph	13.0	12.0	12.0	115.0	12.0	15.0	8.0	15.0	14.0	16.0

Table A.2.: All Durations By Run in seconds (SHIR)