Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Industrie-4.0-Informatik

Bachelorarbeit

von

Simon **Kleber**

LonqAPI: Entwurf und Implementierung einer DaaS Schnittstelle für lang laufende Abfragen in einem AWS Cloud IIoT Data Layer

LonqAPI: Design and Implementation of a DaaS Interface for long-running Queries in an AWS Cloud IIoT Data Layer

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Industrie-4.0-Informatik

Bachelorarbeit

von

Simon **Kleber**

LonqAPI: Entwurf und Implementierung einer DaaS Schnittstelle für lang laufende Abfragen in einem AWS Cloud IIoT Data Layer

LonqAPI: Design and Implementation of a DaaS Interface for long-running Queries in an AWS Cloud IIoT Data Layer

Bearbeitungszeitraum:	von	2. Oktober 2023
	bis	5. Februar 2024

1. Prüfer: Prof. Dr.-Ing. Christoph P. Neumann

2. Prüfer: Prof. Dr. Fabian Brunner

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule
Amberg-Weiden

Selbstständigkeitserklärung

Name und Vorname
des Studenten: **Kleber, Simon**

Studiengang: **Industrie-4.0-Informatik**

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel:

LonqAPI: Entwurf und Implementierung einer DaaS Schnittstelle für lang laufende Abfragen in einem AWS Cloud IIoT Data Layer

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 5. Februar 2024

Unterschrift:

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule
Amberg-Weiden

Bachelorarbeit Zusammenfassung

Student (Name, Vorname):	**Kleber, Simon**
Studiengang:	Industrie-4.0-Informatik
Aufgabensteller, Professor:	Prof. Dr.-Ing. Christoph P. Neumann
Durchgeführt in (Firma):	BHS Corrugated Maschinen- und Anlagenbau GmbH
Betreuer in Firma:	Constantin Ponfick
Ausgabedatum:	2. Oktober 2023
Abgabedatum:	5. Februar 2024

Titel:

LonqAPI: Entwurf und Implementierung einer DaaS Schnittstelle für lang laufende Abfragen in einem AWS Cloud IIoT Data Layer

---

Abstrakt:

IoT Technologien erzeugen große Mengen an Daten, die zur Verbesserung von Produkten und Diensten genutzt werden können. Cloud Computing und Anbieter wie AWS sind entscheidend, um entstehende Big Data Herausforderungen zu lösen. BHS Corrugated, ein führender Anbieter von Lösungen für die Wellpappenindustrie, migriert seine On-Premise Datenmanagement-Infrastruktur in die Cloud, um von den in den Produktionslinien der Kunden gesammelten Daten zu profitieren. Um jedoch Erkenntnisse aus den gespeicherten Daten zu gewinnen, müssen diese zugänglich sein. Derzeit fehlt BHS eine Lösung, um Daten basierend auf lang laufenden Abfragen konsistent bereitzustellen.

In dieser Arbeit wird die LonqAPI vorgestellt, ein Data-as-a-Service Schnittstellenprototyp für lang laufende Abfragen auf dem Data Layer von BHS in der AWS Cloud. Der Schwerpunkt liegt dabei auf Skalierbarkeit, Effizienz und Erweiterbarkeit, um lang laufende Abfragen und Datenquellen weiterzuentwickeln oder zu ändern. Anhand einer Literaturrecherche werden Technologien, Architekturen und Muster für den Entwurf und die Implementierung der LonqAPI identifiziert und verglichen. Das Resultat ist eine REST API, die das Polling Muster für asynchrone Client-Server Kommunikation implementiert und AWS Services wie API Gateway und Lambda für die Schnittstelle und Step Functions State Machines für die entkoppelte Abfrageausführung verwendet. Zur Veranschaulichung der generischen Lösung ist eine Athena-Abfrage als Beispiel für eine lang laufende Abfrage integriert.

Schlüsselwörter: Verarbeitung lang laufender Abfragen, Data-as-a-Service Schnittstelle, asynchrone Client-Server Kommunikation, REST API Polling Muster, AWS cloud-native

Title:

LonqAPI: Design and Implementation of a DaaS Interface for long-running Queries in an AWS Cloud IIoT Data Layer

Abstract:

IoT technologies generate large amounts of data that can be used to improve products and services. Cloud computing and providers like AWS are crucial enablers to solve emerging big data challenges. BHS Corrugated, a leading provider of solutions for the corrugated board industry, is migrating its on-premise data management infrastructure to the cloud to benefit from data collected in customer production lines. However, to gain knowledge from stored data, it must be accessible. Currently, BHS is missing a solution to provide data based on long-running queries in a consistent way.

This thesis presents the LonqAPI, a Data-as-a-Service interface prototype for long-running queries on the Data Layer of BHS in the AWS Cloud. It focuses on scalability, efficiency, and extensibility to further or changing long-running queries and data sources. Through literature research, technologies, architectures, and patterns to design and implement the LonqAPI are identified and compared. The outcome is a REST API implementing the polling pattern for asynchronous client-server communication using AWS services like API Gateway and Lambda for the interface and Step Functions state machines for decoupled query execution. To illustrate the generic solution, an Athena query is integrated as a long-running query example.

KeyWords: long-running query processing, data-as-a-service interface, asynchronous client-server communication, REST API polling pattern, AWS cloud-native

Contents

1 Introduction and Context **1**
1.1 Research Questions 2
1.2 Research Objective 2
1.3 Research Concept 3

2 Fundamentals **4**
2.1 Big Data 4
2.2 Industrial Internet of Things 4
2.3 Cloud Computing 5
2.3.1 Service Models 5
2.3.2 Cloud-native Concepts 5
2.3.3 Amazon Web Services 6
2.4 Data as a Service 6
2.5 Web Service APIs 6
2.6 Long-running Queries 6

3 State of the Art **8**
3.1 Asynchronous Communication 8
3.1.1 Asynchronous Communication Correlation 9
3.1.2 Client-Server Asynchronous Communication Patterns 10
3.2 Query Request Processing 11
3.2.1 Decoupled Invocation Handling 11
3.2.2 Query Processing 12
3.2.3 Large Query Result Handling 13
3.3 API Technologies 13
3.3.1 REST 13
3.3.2 WebSocket 14
3.3.3 GraphQL 14
3.3.4 gRPC 15

4 Case Example **17**
4.1 Introduction to BHS Corrugated 17
4.1.1 Relevance of Big Data at BHS Corrugated 17
4.1.2 Relevance of IIoT at BHS Corrugated 18
4.2 Data Layer at BHS Corrugated 18

4.2.1 Architecture 18
4.2.2 Implementation 19
4.2.3 Relevant Queries and Data Formats 19
4.3 Requirements 20
4.3.1 Functional Requirements 20
4.3.2 Non-Functional Requirements 22

5 LonqAPI Architecture 23
5.1 Architectural Decisions 23
5.1.1 API Technology 23
5.1.2 Asynchronous Communication Pattern for REST 24
5.2 Architecture Components Overview 24
5.3 AWS Service Evaluation 25
5.3.1 REST API Service 26
5.3.2 Query Result Storage Service 26
5.3.3 Query Status Storage Service 27
5.3.4 Query Execution Service 27
5.4 Architecture Overview 31

6 Design Decisions 33
6.1 REST API Design 33
6.1.1 REST Polling Pattern 33
6.1.2 RESTful API Design Practices 34
6.1.3 LonqAPI REST Endpoints 35
6.2 AWS Lambda Request Handling 37
6.2.1 Monolithic vs. single-purpose Lambda Function 37
6.2.2 REST API Python Framework 39
6.3 AWS State Machine Query Handling 42
6.3.1 State Machine Design 42
6.3.2 Data Transformation 44
6.3.3 Result Data 44
6.4 Summary 44

7 Implementation 45
7.1 LonqAPI Monorepo Subprojects 45
7.1.1 mdl-lonq Subproject 45
7.1.2 mdl-daas-history Subproject 46
7.2 mdl-lonq Library Implementation 47
7.2.1 State Machine CDK Constructs 47
7.2.2 Generic Query Processor Lambda Handler 48
7.3 Long-running Query Implementation Process 49
7.3.1 Process Definition 49
7.3.2 Athena POM Query Implementation 50
7.4 mdl-daas-history Project Implementation 51
7.4.1 DaaS History API Infrastructure Stack 52
7.4.2 REST API Lambda Function 53

7.5 Lambda Session Management 54
7.6 Summary 55

8 Evaluation **56**
8.1 API Evaluation 56
8.1.1 Test Client Implementation 56
8.1.2 API Performance Evaluation 57
8.2 Query Execution Infrastructure Efficiency Evaluation 58
8.2.1 Query Execution Time Overhead 58
8.2.2 Query Cost Overhead 59
8.3 Extensibility Evaluation Example 62
8.4 Conclusion 64

9 Summary and Future Work **66**
9.1 Summary 66
9.2 Future Work 66

Bibliography **68**

List of Figures **76**

List of Tables **77**

A LonqAPI **78**

List of Acronyms

API Application Programming Interface
ARN Amazon Resource Name
ASGI Asynchronous Server Gateway Interface
AWS Amazon Web Services
AWS CDK AWS Cloud Development Kit
BHS BHS Corrugated Maschinen- und Anlagenbau GmbH
CRUD Create, Read, Update, Delete
CSV Comma-Separated Values
CTAS Create Table As
DaaS Data as a Service
DML Data Manipulation Language
EC2 Elastic Compute Cloud
ECS Elastic Container Service
EFS Elastic File System
EKS Elastic Kubernetes Service
ELB Elastic Load Balancing
ETL Extract, Transform, Load
GraphQL Graph Query Language
gRPC gRPC Remote Procedure Calls
HATEOAS Hypermedia as the Engine of Application State
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPR Hypertext Transfer Protocol Reliable
HTTPS Hypertext Transfer Protocol Secure
IaaS Infrastructure as a Service
IaC Infrastructure as Code
IAM Identity and Access Management
IIoT Industrial Internet of Things
IoT Internet of Things
JSON JavaScript Object Notation
KMS Key Management Service
MQTT Message Queuing Telemetry Transport
NIST National Institute of Standards and Technology
NoSQL Not only SQL
PaaS Platform as a Service

PDF	Portable Document Format
POM	Point of Measurement
RDS	Relational Database Service
REST	Representational State Transfer
RPC	Remote Procedure Call
S3	Simple Storage Service
SaaS	Software as a Service
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Chapter 1

Introduction and Context

Nowadays, the Internet of Things (IoT) is a ubiquitous technology. With approximately 15.14 billion devices in 2023, it impacts not only our technological world but also how we live and work. Still, its number is expected to increase to more than 29 billion by 2030 [1]. IoT affects various sectors and use cases, from healthcare and agriculture to smart cities and manufacturing [2]. As a subset, Industrial Internet of Things (IIoT) aims to link digital and physical worlds in the context of industrial applications. By enabling data-driven decision-making, remote monitoring, and predictive maintenance, IIoT optimizes processes, increases efficiency, and ultimately improves the bottom line [3].

Along with the promises of IIoT come challenges like security, interoperability, data storage, and data analytics [3]. The volume of data generated can overwhelm existing infrastructure and require new approaches to data storage and processing [4, p. 24]. Cloud technologies are a crucial enabler to solve the challenges of big data generated by IIoT [5, pp. 22-29].

Facing these challenges, BHS Corrugated Maschinen- und Anlagenbau GmbH (BHS), a leading provider of solutions for the corrugated board industry [6], is developing products and services to help their customers benefit from data generated in their production lines. BHS recognized the potential of cloud computing and started migrating its on-premise data management infrastructure to the Amazon Web Services (AWS) cloud in order to build a solid foundation for data-driven innovations and business decisions. However, as stated by Begoli and Horey [7], one principle of effective knowledge discovery on big data is to make data accessible by exposing an Application Programming Interface (API). In order to utilize its full data potential, BHS requires such an interface in its cloud environment. In particular, no solution is available to obtain data from queries that take a long time to execute. Such data access enables numerous use cases, including applications and platforms, data analytics for product improvement, and data as a product usable by customers. To address this, a flexible and extensible solution in the form of a Data as a Service (DaaS) interface must be developed. This thesis aims to facilitate seamless data access and utilization, explicitly targeting long-running queries at the Data Layer at BHS.

1.1 Research Questions

Based on fundamental concepts and definitions described in Chapter 2, this research faces the following problems and aims to answer the associated research questions.

Problem Statement: It is hard to provide data based on long-running queries in a big data environment in a consistent DaaS interface. This challenge is compounded by the multitude of available AWS Cloud services, making it difficult to select and configure suitable infrastructure to efficiently address long-running queries in the cloud. Each storage service has its own data formats, access methods, and performance characteristics. The more data sources are included, the more complex it becomes to implement a flexible and extensible interface. Flexible and extendable in this context means that existing queries can be easily adapted, new queries, data sources, or other services can be integrated, and results are processed and provided in different formats. Moreover, a diverse, changing, and expanding data landscape adds complexity to developing a uniform system handling long-running queries.

Research Question 1 (RQ1): Which technologies and architectures are available to design and implement a DaaS interface providing data from long-running queries in the AWS Cloud?

Research Question 2 (RQ2): How can long-running queries be efficiently processed to provide data based on Athena queries for a DaaS interface using AWS Cloud services?

Research Question 3 (RQ3): How can the system be designed to be flexible and extensible to further or changing long-running queries and data sources?

1.2 Research Objective

The objective of this thesis is to develop a consistent DaaS interface for long-running queries on a Data Layer in the AWS Cloud. The primary goal is to explore challenges and solutions associated with integrating a DaaS approach for data from different sources in a big data environment.

The research aims to achieve effective design and implementation in terms of interface technology and infrastructure architecture. The focus is on developing a reusable and expandable framework concept that can flexibly be applied to comparable scenarios in the AWS Cloud.

By utilizing the Data Layer at BHS as an illustrative example, the research findings will benefit both BHS and other organizations. As a result, these insights will help establish a comprehensive understanding of practical strategies and enable them to adapt and extend the cloud-native DaaS solution concept to meet the current and future needs of internal and external customers.

Cloud computing also poses challenges for security, as presented by Pakmehr et al. [8]. However, neither security aspects nor query optimization are in the scope of this research.

1.3 Research Concept

The research concept entails exploring state-of-the-art literature to identify various technologies, architectures, and design patterns. This exploration is guided not only by the requirements and user stories of the case example at BHS but also encompasses a broader research aspect to answer the defined research questions.

In this context, the *LonqAPI* is developed as a solution to both the case example and the research questions. The name is derived from the terms *long-running* and *query* and refers to the DaaS interface that evolves throughout this research. At BHS, the LonqAPI refers to the DaaS API prototype for long-running queries on historical data. Therefore, design and implementation namings of case example specific components include the terms *daas* and *history* but are part of the LonqAPI.

Once fundamental literature exploration is complete and necessary knowledge is established, the next step focuses on introducing the Data Layer at BHS and its DaaS requirements as the case example on which to apply research results specifically. In order to draw conclusions, the LonqAPI is designed, implemented, and evaluated. Based on a high-level architecture that pays attention to BHS requirements, further research is conducted for design decisions and implementation details. Design and implementation involve generating a concrete solution to handle the Athena POM query as a long-running example described within the case example AWS cloud environment. During development, decisions are made by justifying them concerning previous and further research and considering the requirements.

Throughout and after the implementation of the final software, selected measures are used to determine the quality of the solution. The integration of the LonqAPI as a DaaS API prototype is also evaluated according to BHS's requirements.

Chapter 2

Fundamentals

This chapter introduces the fundamentals necessary to understand the context of this thesis. Divided into six sections, it covers the topics of big data, the Industrial Internet of Things, cloud computing, Data as a Service and web service APIs and defines the term long-running queries.

2.1 Big Data

Sagiroglu and Sinanc [9] define the term Big Data as the combination of the following three V's:

Variety: Data is generated in different formats from different sources. It can be differentiated between structured, semi-structured, and unstructured data.

Volume: The data size is at terabyte or petabyte scale. It challenges traditional storage and analysis techniques.

Velocity: Data can be generated and processed at varying speeds, starting with discrete batches up to (near) real-time or streaming data.

2.2 Industrial Internet of Things

The term Internet of Things (IoT) is described in numerous definitions. In a simple form, IoT refers to a network of interconnected objects that can sense and share information to fulfill a particular purpose [10]. The Industrial Internet of Things (IIoT) is a subset of IoT and references to the use of IoT in an industrial context. IIoT can be characterized by the following aspects to optimize overall production value: networked smart objects, cyber-physical assets, generic information technologies, cloud or edge computing platforms, and real-time enabled processing [11].

2.3 Cloud Computing

The National Institute of Standards and Technology (NIST), a U.S. government entity, defines cloud computing as follows [12]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

2.3.1 Service Models

The NIST outlines three service models for cloud computing that define different levels of abstraction for the customer [12]:

Software as a Service (SaaS): Customers access cloud provider's applications without managing infrastructure. Applications are available via web browsers or program interfaces.

Platform as a Service (PaaS): Customers deploy their own applications using supported tools without handling infrastructure. They control deployed applications and hosting configuration settings.

Infrastructure as a Service (IaaS): Customers provision computing resources to run arbitrary software. They do not manage the cloud infrastructure but control fundamental computing resources like storage, processing, and network.

2.3.2 Cloud-native Concepts

Cloud-native computing is an approach to building and running applications based on the cloud computing paradigm and utilizing its advantages. A definition is provided by Gannon et al. [13], and the most important concepts are summarized in the following.

Cloud-native applications operate globally distributed, are scalable, fault-tolerant, and seamlessly upgrade- and testable. Security is integrated into the application and its infrastructure. Microservices, containers, and their orchestration are common paradigms in cloud-native computing. A microservice is small, independent, and loosely coupled. They are often packaged in containers, isolated environments running applications with less overhead than virtual machines. Orchestrated by tools like Kubernetes, containers can be deployed, scaled, and managed independently and automatically.

However, cloud-native computing is not limited to these concepts. Cloud providers offer a broad set of services to offer new capabilities for applications. Serverless computing is an essential concept in this context. It allows running code on-demand without provisioning or managing servers. Other examples of managed services cover storage, messaging, or machine learning.

2.3.3 Amazon Web Services

Amazon Web Services is a public cloud provider offering a broad set of over 200 services. The company describes itself as “a highly reliable, scalable, low-cost infrastructure platform in the cloud” [14]. Next to Microsoft Azure, Alibaba Cloud, and Google Cloud Platform, AWS is one of the major cloud providers and has the biggest share with 40 % in the public cloud IaaS market worldwide in 2022 [15].

2.4 Data as a Service

Like SaaS, Data as a Service (DaaS) is a service model for cloud computing. Its goal is to provide data in a standardized format on demand to consumers irrespective of geographical or organizational separation between provider and consumer. In contrast to common data access within an organization that enables all methods of Create, Read, Update, Delete (CRUD) operations, DaaS, in general, is limited to read-only access. DaaS decouples data from applications, reducing maintenance expenses, enhancing services, and increasing data value [16].

2.5 Web Service APIs

Web services are open Internet-oriented applications that enable interoperability between heterogeneous systems. They rely on standard protocols, including the Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP), to exchange data in common formats like Extensible Markup Language (XML) or JavaScript Object Notation (JSON) [17, Sec. 5.1.1] [18].

Web services can implement APIs, also called web service APIs or web APIs [19]. The term API stands for *Application Programming Interface* and describes software interfaces that provide access to data or services for internal or external consumers. In contrast to websites, APIs ensure structure in their use and content. This concept can be seen as a contract between the provider and the consumer of the API to enhance efficiency through documentation, consistency, and predictability [20, pp. 4-5].

2.6 Long-running Queries

As discussed in [21], queries on very large data can be a challenge for database administrators. A query execution time can be unpredictable and may take seconds to hours to complete. Depending on the context, the term long-running can be defined differently. For real-time applications in embedded systems, a response time within 10 milliseconds can be required [22, pp. 5-6]. Processes that require computing resources on the scale of supercomputers can take days to weeks to complete, and operations within minutes are considered short [23]. Interfaces that are used by humans should respond within a few seconds, as stated by Podelko [24]. Operations that take longer than 10 seconds should be specially treated by the user interface. In the context of the LonqAPI, data can be

provided to such interfaces. Based on these considerations, the term long-running in this thesis refers to queries that can not guarantee a response time within ten seconds.

Chapter 3

State of the Art

This chapter summarizes existing approaches for long-running operations on big data and how they can be handled by asynchronous communication. It also gives an overview of popular API technologies and compares their main characteristics. The outcomes discussed here are used to design a high-level architecture for the LonqAPI in Chapter 5.

3.1 Asynchronous Communication

The goal of this thesis is to provide a solution to enable clients of the LonqAPI to retrieve data based on long-running queries. The main problem is that the client has to wait for the data to be available. Within synchronous communication, the client task is blocked until it receives a response containing the necessary data. This is sufficient for short operations but not for long-running queries. It is not a good user experience, especially if the query can take several minutes to complete [25]. Brambilla et al. [26] show that these problems can be solved using an asynchronous communication approach. The client can send the request and continue its execution while the query is processed. Results can be retrieved once they are available. This way, asynchronous interactions ensure a consistent response time for the client. Figure 3.1 outlines the general difference between synchronous and asynchronous communication for the client.

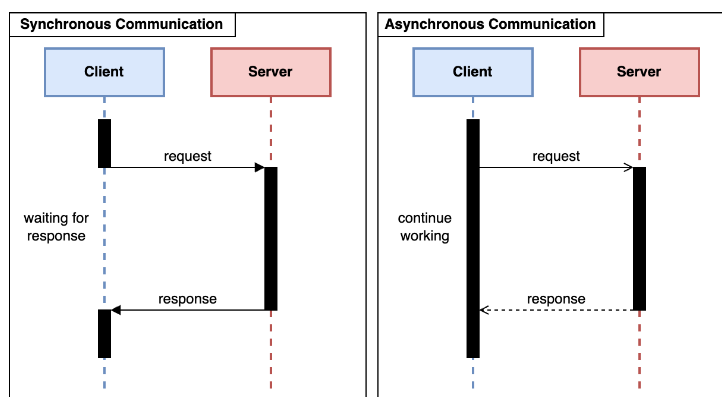

Figure 3.1: Synchronous vs. asynchronous communication (adapted from [27])

Asynchronous interactions add complexity to the web service architecture. The API server must face the following two main challenges [26]:

1. Long persistence of the request context
2. Logical request and response correlation

In case of the LonqAPI, request context refers to the status of the query execution and the resulting data. The client can retrieve the status and results later in time, providing correlation to the original request.

3.1.1 Asynchronous Communication Correlation

Brambilla et al. [26] describe three methods for correlation in asynchronous communication.

Correlation at Transport Level

Correlation at transport level is defined as the set of communication protocols that provide a mechanism to correlate requests and responses. In 2004, Brambilla et al. mentioned that protocols exist that natively support asynchronous communication. An example cited is Hypertext Transfer Protocol Reliable (HTTPR), a protocol based on HTTP that encapsulates application metadata in the payload [28]. However, HTTPR is not widely adopted and has never become a standard since its introduction in 2001 [29].

Today, there are other protocols that enable asynchronous interactions. For instance, WebSocket or Message Queuing Telemetry Transport (MQTT) are more widely adopted and can be used to implement asynchronous communication. With both protocols, the client can keep the connection open and receive messages from the server at any time [30, p. 9], [31]. Relevant protocols and frameworks are discussed in more detail in Section 3.3.

Correlation by Application Semantics

Correlation by application semantics uses the application data structure to embed correlation information. For example, the data could be a business document, like a report Portable Document Format (PDF), and the correlation information could be a reference number within the document, like the report ID on the first PDF page. The relation between requests and responses is implicit. This method is more complex and less light in implementation as the transport level correlation. Its advantage is that it can be used with synchronous protocols.

This method has the following disadvantages according to Brambilla et al.:

1. Application data structure has to be shared between client and server.
2. Unsuitable application data formats have to be converted.
3. Application data and implementation issues are not separated.

Correlation by Conversation Metadata

Conversation metadata is separate information next to the application data. It describes the interaction between client and server and can be used to correlate requests and responses. The correlation can be on conversation level, relevant to multiple service calls, or on operation level, used to relate a single request and response. Both levels use a unique identifier, conversion ID, or operation ID for correlation. This identifier is included in the metadata of the request and response messages, separated from the application data. For example, the application data can be an email message, and the correlation identifier can be part of the email metadata as the subject. Like correlation by application semantics, this method also can be used with synchronous protocols but solves the disadvantages mentioned in 3.1.1.

3.1.2 Client-Server Asynchronous Communication Patterns

Brambilla et al. [26] demonstrate that asynchronous communication can be implemented using different patterns with a synchronous protocol and a correlation identifier. This section summarizes these and other patterns based on point-to-point communication, which can realize asynchronous communication in a client-server web service architecture. Polling and callback in this context are also described by Arulanthu et al. [32] and Voelter et al. [33]. Voelter et al. additionally present the fire and forget and sync with server patterns, in which the client does not receive results of the asynchronous operation. Eugster et al. [34] discuss the publish/subscribe pattern in more detail.

Besides client-server communication, asynchronous communication can also be implemented in a peer-to-peer architecture. In contrast to the client-server approach, participants in peer-to-peer communication share resources like computing power with other peers. This is not applicable to the LonqAPI as a centralized service providing data and will not be carried out further in this thesis. A definition of peer-to-peer networking is discussed in more detail by Schollmeier [35].

Fire and Forget

The fire and forget pattern is the simplest asynchronous communication pattern and is realized with a single one-way message. The client sends a request to the server to start an operation and does not expect a response. The client does not know if the server received the request or what the status or result of the operation is [33].

Sync with Server

The sync with server pattern is similar to the fire and forget pattern. The client sends a request to the server to start an operation but expects a reply to the reception of the request. The server starts the requested operation independently and does not answer more than the reception confirmation to the client. The client knows the server received the request but not the status or result of the operation [33].

Polling

The polling pattern is based on synchronous two-way request-response communication. The client sends a request to the server to start an operation and receives a response with a correlation identifier. The client has no information at which point in time the result is available but can poll for it by sending a request with the correlation identifier. He may poll for the result when it is not available and has to retry later. Once the operation is completed and the client polls for it, the server sends the result within the response. This is inefficient, can lead to more requests than necessary, and the client does not receive the result as soon as it is available. On the contrary, the polling pattern is easy to implement and requires no special service from the client [26], [32], [33].

Callback

A callback is a particular service provided by the client. The client sends a request to the server and provides the callback address. The correlation identifier is either provided by the client or generated by the server and returned in the acknowledgment. The server handles the operation and sends the result to the callback address once available. This pattern is more efficient than polling because no unnecessary requests are sent, and the client receives the result as soon as it is available. It makes communication more flexible because the result receiver can be different from the request sender. However, the client must provide a service itself that is compatible with the server message [26], [32], [33].

Publish/Subscribe

The publish/subscribe pattern is similar to the callback pattern and can optionally include acknowledgments. The client subscribes to a topic and sends a request to the server. The server handles the operation and publishes the result to the topic. Unlike the callback pattern, multiple callback messages can be sent from the server. This can be used for repetitive information and event notification or to send the result in multiple parts. The publish/subscribe pattern can also be used for one-to-many communication, where multiple clients subscribe to the same topic [26], [34].

3.2 Query Request Processing

Section 3.1 described the communication between the client and the server. This section focuses on processing of the query request on the server side, including invocation management, query processing, and large query result handling.

3.2.1 Decoupled Invocation Handling

Synchronous requests can be handled and run by the server within one execution thread. This is possible because the client is blocked until the operation is completed and the result is returned as a response. Asynchronous requests need to be treated differently. The client expects a response immediately after sending the request, even though the

operation has not been completed yet. The server has to handle the request and run the operation decoupled from the invocation in a separate thread or process [25], [36, pp. 215-218].

One method to decouple invocation dispatching and handling is to use a queue to buffer the asynchronous operations, as mentioned by Zdun et al. [25] and described by Richards [37, pp. 11-14] and Sbarski et al. [38, Sec. 3.2.3]. Requests as events are put into the queue by one or multiple dispatcher processes. The requests stay in the queue until a worker process is available to handle a request. An event mediator can be used in between to allocate the events in the queue to available workers. Each worker can process multiple requests after each other. Figure 3.2 shows the described workflow. Queues in event-driven architectures enable separation of concerns and loose coupling between components, making the system more scalable and robust.

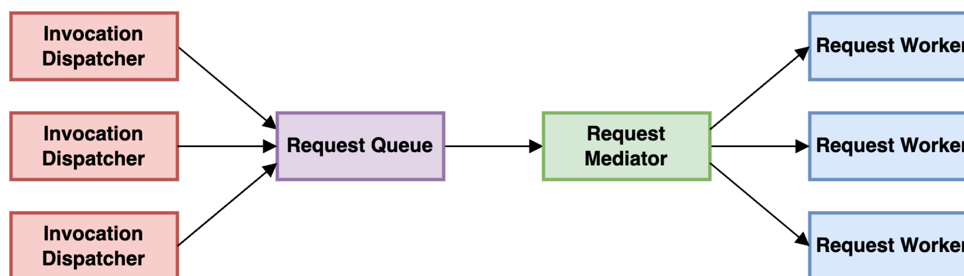

Figure 3.2: Decoupled request invocation handling workflow using a queue with a mediator and multiple request workers (adapted from [38, p. 48])

3.2.2 Query Processing

The query initiated by a client request is executed using a worker process. The goal is to provide the desired result easy to use for the client. This process depends on the specific use case but can be divided into multiple consecutive steps.

Extract, Transform, Load (ETL) processes are used to extract data from different sources, transform it, and load it into a target, usually a data warehouse. The data warehouse makes the collected data available for analysis and reporting. In the case of the LonqAPI, the target data storage does not have to be a data warehouse, but the steps can be applied to the query processing. The following paragraphs describe the three ETL stages and how they can be applied to query processing. Figure 3.3 shows the ETL-inspired stages in the context of LonqAPI query processing [39], [40].

Extraction: The extraction step retrieves data from source systems and handles their heterogeneity. The data can be stored in an intermediate storage for further processing, also called a staging area. For requests to the LonqAPI, this step can be used to perform the actual query on the cloud storage services and to extract the data for further processing.

Transformation: The transformation step can include cleaning, transforming, and integrating the data. The purpose is to correct, complete, and bring the data into a

consistent schema for storage in the target system. This step represents the use case specific changes on the data provided by the LonqAPI.

Loading: The loading step stores the data in the target system, which, in general, is a data warehouse. This step results in the data stored in the target system, ready to be used by the end-user. In the case of the LonqAPI, the result is the data provided to the client, which he may access directly or via API calls.

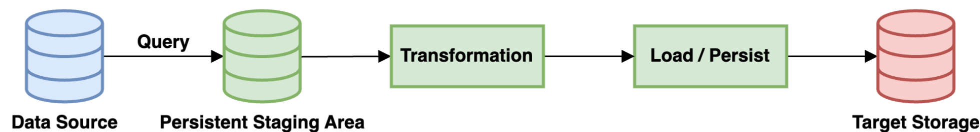

Figure 3.3: The ETL-inspired query processing stages in the context of the LonqAPI

3.2.3 Large Query Result Handling

After query execution, the result has to be processed until it is ready for the end-user. Depending on the query and its output, the result faces different challenges inherited from its big data sources. To handle these challenges, the following principles can be applied [41, p. 21]:

1. The result can be split into multiple parts to reduce the processed volume.
2. Multiple results can be processed parallel to reduce overall processing time.
3. Results should be processed from any point of failure to avoid restarting the whole process.
4. Results should be processed consistently to avoid differences in format, type, or structure.
5. Many algorithms need to be applied to process the data quickly and efficiently.

3.3 API Technologies

The following sections overview popular API technologies, including REST, WebSocket, GraphQL, and gRPC. A comparison of the technologies is presented in the context of the use case of the LonqAPI. The comparison also includes an evaluation of the compatibility with currently available services in the AWS Cloud. Other considered characteristics are the support of asynchronous communication at transport level, maturity, and usability for client and server in terms of implementation for a DaaS API.

3.3.1 REST

REST stands for "Representational State Transfer" and is an architectural style for API design. It is a stateless client-server architecture initially introduced by Roy Fielding in [42]. The core abstraction used in a REST API is the concept of resources. Fielding defines a resource as "any information that can be named". Based on the HTTP protocol, clients can use its methods like GET, POST, PUT, and DELETE to perform CRUD operations on resources provided by the server. A request from the client or

the response from the server may contain control data to define the purpose of the message or to parameterize the request. These resources are identified by a Uniform Resource Identifier (URI) and can be represented in different formats like XML, JSON, or Hypertext Markup Language (HTML) [43, p. 54]. The media type, or the last-modified time, are examples of metadata that can be provided to describe a representation of a resource. Response messages can also contain information that is not specific to the representation but is metadata to the resource itself, like source links [44].

REST aims to reduce network communication and latency while increasing the independence and scalability of components [44]. REST and HTTP do not support asynchronous communication at transport level [26]. In combination with the REST architectural style, HTTP is currently the most widely used API technology [45]. Many frameworks, programming languages, and tools support it, which makes it easy to implement client and server applications [46]–[48]. AWS provides a managed API service called API Gateway that enables a REST architecture and integrates with other AWS services like AWS Lambda, Elastic Compute Cloud (EC2), or DynamoDB [49].

3.3.2 WebSocket

In contrast to HTTP, the WebSocket protocol enables stateful, bidirectional, full-duplex communication between client and server. After an initial HTTP handshake, the connection is upgraded to WebSocket based on the Transmission Control Protocol (TCP). The connection stays open until it is closed by the client or the server and can be used to send messages in both directions at any time. REST requires a new TCP connection for each request. A WebSocket connection uses only one TCP connection and can transmit multiple messages [50].

The WebSocket protocol provides a standard for asynchronous communication at transport level [30, p. 9]. It is more efficient for frequent messages and can reduce latency compared to HTTP by lower overhead per message [50]. Not as widely adopted as HTTP, nowadays almost all browsers support WebSocket [51], and multiple libraries are available for client and server applications [52]. In addition to REST via HTTP, the AWS API Gateway service supports WebSocket APIs [49].

3.3.3 GraphQL

GraphQL is a query language for APIs and a server-side runtime for fulfilling queries with existing data. Initially developed by Facebook in 2012, it was open-sourced in 2015 and is now maintained by the GraphQL Foundation since 2019 [53]. Typically served over HTTP, a GraphQL API exposes a single endpoint for all provided capabilities. The API server defines a schema, including types, fields in these types, and operations that can be performed on these types and fields. Relations between types and fields define a graph that gives GraphQL its name. Operations are queries, mutations, and subscriptions. A query is used to retrieve data, a mutation to change data, and a subscription to receive data changes in real-time. On the server-side, each field on each type is backed by a resolver, a function responsible for returning that field's value. These

resolvers need to be implemented by the developer and can be used to retrieve data from any source. The exchanged data is, in general, in the JSON format. In contrast to queries and mutations, subscriptions can maintain an open connection, enabling the server to push data to the client. To allow subscriptions, the server must support a protocol to maintain the connection. Therefore, the most commonly used is the WebSocket protocol [54], [55].

GraphQL can be more efficient than REST because it allows clients to request only the data they need, making the API more flexible. It reduces the problem of over- and underfetching data that is common in REST APIs with endpoints that return fixed data structures. Related to underfetching, GraphQL can reduce the number of requests needed to retrieve the desired data. The schema helps the server to validate and execute operations and provides the client with structured information to understand the capabilities of the API. Compared to REST, GraphQL is a relatively new technology that is less widely adopted and can require developers to learn it. However, many libraries are available for GraphQL client and server applications [56]. If the GraphQL Server supports subscriptions, it can be used to implement asynchronous communication at transport level. Like API Gateway for REST and WebSocket, AWS provides a managed service for GraphQL called AWS AppSync [57]. A more detailed comparison of GraphQL and REST is provided by Vadlamani et al. [45].

3.3.4 gRPC

Remote Procedure Call (RPC) is a communication protocol that enables a client to call a procedure on a remote server [58, p. 39-40]. gRPC Remote Procedure Calls (gRPC) is one of the most popular RPC frameworks, developed by Google and open-sourced in 2015 [59]. It is based on HTTP/2 and, by default, uses protocol buffers as its interface description language and for data serialization. Services are defined in a proto file, an ordinary text file. This file defines the methods that can be called remotely together with their parameters and return types. For supported languages, including C++, Java, Python, and Go, the gRPC framework can generate client and server code from the proto file. Generated code simplifies data access through typed methods and data structures or by enabling serialization and parsing of binary data. A server can implement multiple services and expose them to handle client requests. The gRPC infrastructure handles the communication between client and server, including serialization and deserialization of data, connection management, and error handling. A client can call a remote procedure on the server by invoking a method on a stub representing the remote procedure. The stub is a client-side object that represents the server as if it were a local object. A channel with specified arguments defines the connection between a stub and the server. Service parameters and return types can be defined as streams to provide data in sequences. Methods can be called synchronously or, depending on the programming language, asynchronously [60], [61].

gRPC allows the developer to focus on the application logic and ensures abstraction from the underlying client-server communication. The binary transfer format via HTTP/2 can be more efficient than other text-based formats like JSON used by REST or GraphQL.

gRPC enforces clear and well-defined service interfaces between client and server, supporting development experience and maintainability. Native streaming support extends the request-response model and can transfer extensive data in sequences. If the gRPC implementation supports asynchronous service calls, it can be used for asynchronous communication at transport level. The framework supports multiple programming languages [60]. However, gRPC has a relatively small ecosystem and less developer popularity than REST. The browse support is limited, which can be a problem for client applications. Service definition changes require client and server code changes, and the strongly typed characteristics may be a disadvantage for client flexibility. AWS does not provide a managed service built to support gRPC, but Amazon EC2 instances can be used to host gRPC services. The Elastic Kubernetes Service (EKS) and Elastic Load Balancing (ELB) can be integrated to orchestrate and load balance them [62]. Comparisons of gRPC with other API technologies are presented by Indrasiri and Kuruppu [63, Ch. 1] and Buzhin et al. [61].

Chapter 4

Case Example

This chapter introduces the case example that is used throughout this thesis. It is based on a real-world project at BHS and illustrates the concepts and methods presented. The first section of this chapter briefly introduces the company BHS, focusing on the relevance of IIoT and big data. The second section describes the existing Data Layer at BHS and provides information to understand the context of this thesis. The third section defines the requirements of BHS for the LonqAPI as its DaaS interface prototype.

4.1 Introduction to BHS Corrugated

BHS Corrugated Maschinen- und Anlagenbau GmbH (BHS) is the largest solution provider in the corrugated board industry [6], with 20 locations worldwide and over 3000 employees. Headquartered in Weiherhammer, Germany, BHS is primarily involved in the construction, maintenance, and new developments of corrugator machine groups. In the context of maintenance, BHS provides its customers with various digital products, among others.

4.1.1 Relevance of Big Data at BHS Corrugated

Digital solutions at BHS are mainly based on collected production data from various systems, primarily in customer plants. In more than 200 connected corrugators, the production data of approximately 3,500 measured data points are gathered, validated, and processed. The near real-time to real-time data uses storage space at a terabyte-scale in on-premise data stores. Cloud-based systems are not yet production-ready but are in development, as described in section 4.2. Regarding the three V's of big data as outlined in section 2.1 on page 4, BHS and its solutions can be classified in a big data environment.

4.1.2 Relevance of IIoT at BHS Corrugated

As mentioned in the previous section, a large share of data is collected from connected corrugators. This data is mainly produced by sensors, actuators, and operational systems. Gathered by servers in the customer plants, the data is transferred to BHS's on-premise and cloud-based systems. Regarding the aspects mentioned in section 2.2 on page 4, BHS and its solutions can be classified in an IIoT environment.

4.2 Data Layer at BHS Corrugated

To validate, store, and provide the data collected from different sources at BHS, a Data Layer in the AWS cloud is being developed. It is designed to replace on-premise data storage and processing infrastructure in the future. Existing data is stored in relational databases and a key-value Cassandra database cluster. This on-premise infrastructure is limited in its scalability and flexibility and causes regular maintenance efforts and costs. The Data Layer in the cloud aims to solve these problems by providing a central data infrastructure based on managed AWS services. In its current state of development, the Data Layer is able to receive and store specific data. Some other features and components are planned and scheduled to be implemented in the future. One of these components is the DaaS interface for historical data, which prototype is developed in the context of this thesis. The following describes the current architecture and implementation of the Data Layer, limiting ourselves to the components and details necessary for this work. Components not relevant in this context are data ingestion and validation, a live data interface, and a legacy systems interface.

4.2.1 Architecture

The Data Layer is developed as a cloud-native application in the AWS cloud, using different services to process, store, and provide data. Collected from multiple IIoT sources, the data of various types are transferred to the Data Layer in a consistent way. Only queries and data pertinent to the LonqAPI will be presented in the following. By ignoring the data ingestion, we can assume that the data is already validated, optionally transformed, and in a format that is ready to be stored and queried.

In its current state of development, the Data Layer stores data in two different storage services: Amazon Simple Storage Service (S3) and DynamoDB. Not yet in use, the Relational Database Service (RDS) service is planned to be integrated. Figure 4.1 shows the simplified architecture of the Data Layer. Depending on the data's type, format, and usage, it is stored in one of the three services.

S3: The S3 service is an object storage service [64], which is used in this case to store data in the format of parquet files.

RDS: The RDS service is a managed database service to store data in a relational database [65].

DynamoDB: The DynamoDB service is a fully managed NoSQL database service that stores data in a key-value format [66].

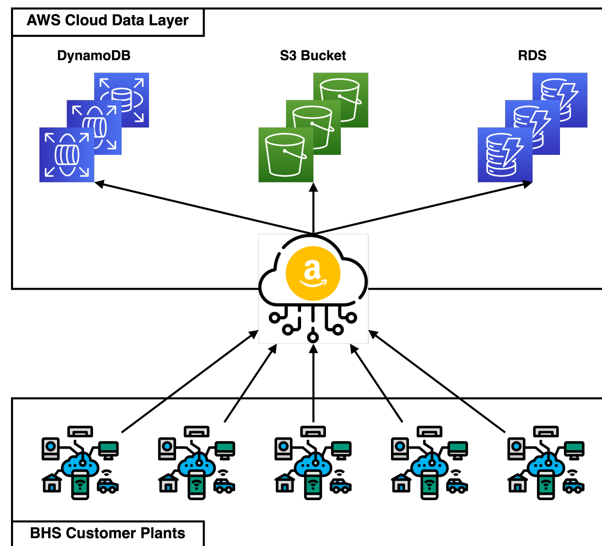

Figure 4.1: The simplified architektur of the Data Layer at BHS Corrugated

4.2.2 Implementation

The Data Layer uses the AWS Cloud Development Kit (AWS CDK), an Infrastructure as Code (IaC) software development framework, to define, deploy, and manage AWS cloud infrastructure in code [67]. Most of the relevant software for this thesis is written in Python, and so is the AWS CDK Python library used to define all infrastructure.

All Data Layer code is organized in a monorepo, "a single repository that contains multiple related or unrelated projects" [68]. One of these projects is *mdl_cdk_core*, which is represented as a Python module within the repository containing all fundamental infrastructure components of the Data Layer, including storage services.

Another module is *mdl_data_access*, which aims to provide an interface implementation to access data stored in the Data Layer. It is an abstraction layer to access data from different storage services and a utility library for format conversions. Relevant in the development of the LonqAPI is one Athena query implementation, which is adapted from the *mdl_data_access* module. Further information regarding relevant queries and data formats can be found in subsection 4.2.3.

The Pants build system [69] is used to manage dependencies, build, and test all projects in the monorepo. Combined with existing Bash scripts and AWS CDK commands, Pants is also used to deploy the Data Layer infrastructure to the AWS cloud.

4.2.3 Relevant Queries and Data Formats

The Data Layer stores data from different sources in different formats. The most significant amount of data is produced by various sources at the customer corrugators,

including sensors and actors. It is named *Point of Measurement* (POM) data and is stored in S3 buckets in parquet files. Apache Parquet is a column-oriented data file format. It utilizes compression and encoding schemes "to handle complex data in bulk" [70]. The POM data can be queried using Structured Query Language (SQL) statements via the AWS Athena service. Athena is a query service that uses standard SQL to analyze data stored in S3 buckets. For Athena to be able to query data in S3, an AWS Glue Data Catalog is required as reference to the stored data [71]. Such a catalog is already available for the POM data and can be used with the SQL query.

Relevant to this thesis is an Athena query to retrieve POM data from a single corrugator specified by its area ID. The query filters the data by POM IDs and a time range with timestamps in milliseconds since the Unix epoch [72, Sec. 4.16]. To reduce the amount of scanned data, the WHERE clause also filters by year, month, day, and hour, which are used as partition keys in the S3 bucket. This query statement is documented in listing A.1, and an example result is shown in figure A.1 in the appendix. In the context of this thesis, the query is referred to as the Athena POM query.

POM data values are stored in the metric system, but some applications require data in imperial units. For that reason, the *mdl_data_access* module provides a utility function to convert POM data from metric to imperial units. Its input is a *pandas.DataFrame* [73] containing a POM ID and a value column. The function requires access to a mapping DynamoDB table that contains the POM IDs and their corresponding transformation calculations. The header of this function is shown in listing A.2. Even so, the transformation calculations could be implemented in the client application using the data, this is implemented in the Data Layer for legacy applications at BHS and is therefore included in the requirements in the following section.

4.3 Requirements

This section defines the functional and non-functional requirements of BHS for the LonqAPI as a DaaS interface prototype in its Data Layer.

4.3.1 Functional Requirements

The functional requirements are defined as user stories. Therefore, the following structure is used as suggested by Cohn [74, pp. 80-81]:

> I, as a [role], want [function] so that [business value].

The user stories aim to fulfill the quality attributes of the INVEST acronym [74, Ch. 2]: Independent, Negotiable, Valuable, Estimatable, Small and Testable. Each story includes a list of acceptance criteria to verify the implementation.

The following user stories are defined for the LonqAPI:

1. I, as a developer, want to retrieve filtered historical POM data from a single production plant so that I can integrate it into new and existing applications.

(a) A necessary request parameter is implemented to identify the API client by its client ID.
(b) A necessary request parameter is implemented to identify the production facility of a client by its area ID.
(c) Necessary request parameters are implemented to filter the data by a list of POM IDs.
(d) Necessary request parameter timestamps in milliseconds since the Unix epoch [72, Sec. 4.16] are implemented to filter the data by a time range.
(e) An optional request parameter is implemented to convert the data from metric to imperial units with a default to metric units.
(f) The data is retrieved from the Data Layer using the defined Athena POM query in listing A.1.
(g) If imperial units are requested, the data is converted before it is returned using the existing conversion function in listing A.2.
(h) Non-functional requirements are evaluated and fulfilled.
(i) Tests have been performed in the Data Layer development environment.

2. I, as a developer, want to retrieve an error message if I provide invalid request parameters so that I can correct my request and avoid failures in my application.
   (a) An error code is implemented to indicate that the request parameters are invalid.
   (b) A message is returned to tell the user which request parameters are invalid.
   (c) No query is executed.
   (d) No data is returned.
   (e) Non-functional requirements are evaluated and fulfilled.
   (f) Tests have been performed in the Data Layer development environment.

3. I, as a developer, want to retrieve an error message if querying the data fails so that I can implement a fallback in my application.
   (a) A status code is implemented to indicate that the query failed.
   (b) A message is returned to explain the status code.
   (c) No data is returned.
   (d) Non-functional requirements are evaluated and fulfilled.
   (e) Tests have been performed in the Data Layer development environment.

4. I, as an external developer, want to retrieve API documentation to understand how to use the DaaS interface so that I can reduce the time to integrate the interface into my application.
   (a) All available data query requests are included in the API documentation.
   (b) Each request is documented with a description according to its parameters.
   (c) Each request is documented with a description according to its return values.
   (d) Each request is documented with a description according to its error codes.
   (e) Non-functional requirements are evaluated and fulfilled.
   (f) Tests have been performed in the Data Layer development environment.

4.3.2 Non-Functional Requirements

Explicitly not included in the requirements in the context of this thesis are the following aspects:

1. Optimization of query execution performance.
2. Special precautions regarding API security and data protection.

The following non-functional requirements are defined for the LonqAPI:

1. The naming of the API apart from the domain name includes the terms *DaaS* and *history* to indicate that the API provides historical data.
2. The system has to be designed to handle long-running queries with execution times up to one day.
3. The system has to be designed to handle large data query results of up to 10 GB.
4. New queries can be integrated and existing queries can be modified without changing the system's principal structure.
5. New data sources can be integrated without changing the system's principal structure, including Amazon RDS and DynamoDB.
6. The API must support industry-standard technologies and formats to ensure compatibility with various applications and tools.
7. Query results have to be available to the user for at least one day after the query execution.
8. The system can scale horizontally to accommodate increased user load of up to 10000 requests per minute (not including long-running queries).
9. The system does not manipulate the original data value except for format or unit conversions if requested.
10. The primary programming language of the system should be Python at version 3.10.
11. The system uses the existing Pants build system structure for all modules.
12. All implemented packages follow the Data Layer convention by starting with *mdl-*.
13. The system is deployed using the AWS CDK framework and infrastructure stacks are implemented within the *mdl-cdk-core* package.
14. Lambda functions are deployed as Docker containers and use the x86 architecture runtime.
15. Unique IDs used in the system are in Universally Unique Identifier (UUID) format of version 4 [75].
16. Timestamps transmitted to the client are in ISO 8601 format [76].

Chapter 5

LonqAPI Architecture

This chapter constructs the architecture of the LonqAPI in the AWS cloud based on research results in chapter 3 and requirements in chapter 4. The first section presents decisions that influence the architecture, and the second section describes abstract components necessary within the architecture. The third section replaces these abstract components with concrete AWS services after evaluating possible solutions. The last section describes the resulting architecture of the LonqAPI.

5.1 Architectural Decisions

This section describes decisions based on research results in chapter 3 and requirements for the LonqAPI in chapter 4. The decisions include the selection of an API technology and an asynchronous communication pattern for it.

5.1.1 API Technology

The API technologies considered are REST, WebSocket, GraphQL, and gRPC. They were described and compared in section 3.3. Every technology would be a possible solution for API implementations, but not all of them are suitable for the LonqAPI and its requirements outlined in section 4.3.

The best suited for the LonqAPI is REST. It is the most common API technology and, therefore, the most familiar to developers. Newer technologies like GraphQL and gRPC are less established than REST, especially for mechanical engineering companies like BHS, which are not focused on software development. REST does not support asynchronous communication at transport level, but other correlation methods can be implemented, as highlighted in section 3.1.1. WebSocket supports asynchronous communication at transport level but would require the client to keep a connection open until the query is finished, which is not suitable for operations that can take several minutes to hours. GraphQL is more flexible than REST, offers a subscription mechanism, which could be used to subscribe to query results, and can be more efficient than REST by transferring only the data requested by the client. However, for the use case of the

LonqAPI, the flexibility of GraphQL is less relevant, because the API resource of a query and its results are simple and do not require a complex API query language. gRPC can be a good choice for internal APIs but is unsuitable for public APIs because it is not as widely supported as REST. AWS also does not offer a service for gRPC APIs, as it does for the other technologies.

5.1.2 Asynchronous Communication Pattern for REST

As described in section 3.1.2, there are five patterns to implement asynchronous communication: fire and forget, sync with server, polling, callback, and publish/subscribe. Fire and forget and sync with server are not suitable because the client does not receive the results of the operation. Compared to the other two patterns, the polling pattern is the simplest to implement and, therefore, the best choice for the LonqAPI. It does not require the client to provide a special service as the other two patterns do. The drawback is that more requests can be necessary to retrieve the result of a query that the client does not receive immediately. Callback and publish/subscribe would be more efficient but would make the API usage for a client more complex, and if the client could not provide a suitable service to call, he would not be able to retrieve the results by himself.

5.2 Architecture Components Overview

Research results in chapter 3, requirements for the LonqAPI in chapter 4, and architectural decisions in section 5.1 construct the required components for the LonqAPI architecture. This section describes these components, their requirements, purpose, and interaction with each other, illustrated in figure 5.1.

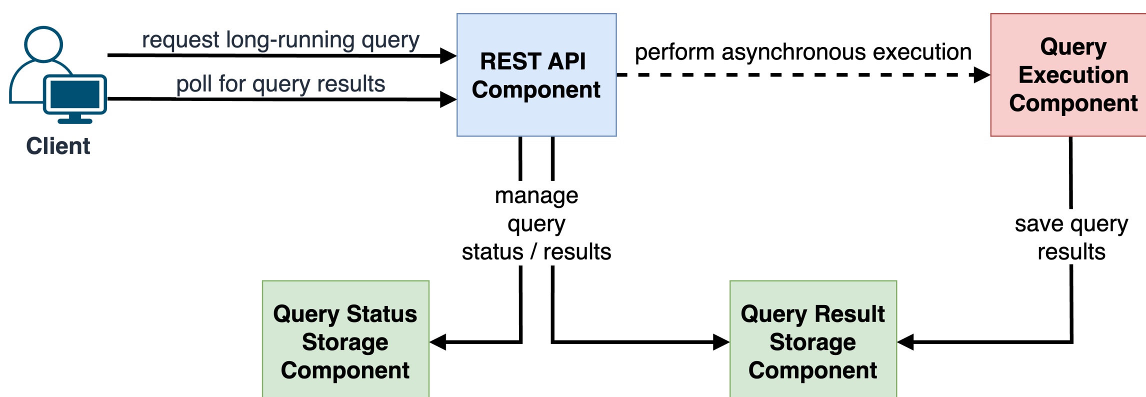

Figure 5.1: Overview of the required LonqAPI architecture components and their interaction

REST API Component: The first component is an API service that provides an interface for clients to send requests to the LonqAPI. As evaluated in section 5.1, the API technology used for the LonqAPI is REST, and the pattern to implement asynchronous communication is polling. Its main purpose is to handle client requests, invoke the query execution asynchronously, and return a synchronous response to the client. This component interacts with the query execution component to invoke the query processing asynchronously and with the query result storage component to provide the data to the

client. The status of a query result is stored in the query status storage component and needs to be retrieved by the API service to return it to the client in response. To reduce latency, the REST API service should be highly available, scalable, and performant.

As a server instance, the REST API component could also include the following storage components, but they are separated to reduce coupling within the architecture. This facilitates architecture changes, and deployments of the API service do not affect the persistence.

Query Result Storage Component: The query result storage component within the LonqAPI architecture stores the processed results of long-running queries. As described in section 3.2.2, the query result processing can be compared to an ETL process. The Loading step is the last in the ETL process and stores the results ready for the client to retrieve. Therefore, the query result storage component needs to be able to store large amounts of data and provide a way to retrieve it. Regarding the requirements of the LonqAPI defined in section 4.3, results can be up to 10GB in size, must be available for at least one day, and must be in industry-standard formats. To reduce the processed volume of data, the results can be split into multiple parts. The storage should also not be limited to query results of a specific data source in order to design a consistent architecture for the LonqAPI.

Query Status Storage Component: As stated in section 3.1, two challenges of asynchronous communication are the correlation of requests and responses and the persistence of the request context. This request context refers to the query request's status and results. The client needs to be able to retrieve both based on the correlation identifier of the request. Therefore, the query status storage component is necessary to persist the request status and implementation-specific context. This service does not need to store large amounts of data, but it should be highly available and performant to reduce the latency of the LonqAPI on client polling requests. To reduce the coupling between architecture components, the query status is only managed by the REST API component. Another possibility would be to update the status by the query execution component on changes.

Query Execution Component: The REST API component is in charge of the client request invocation, and the query results are persisted in the query result storage component As described in section 3.2, the query defined by a client request must be decoupled from its invocation, and results must be processed into a format usable by the client. The REST API service used as the invocation dispatcher has to initiate a worker service to execute and process the query and store the results while the API service returns the response to the client. This worker service is the query execution component of the LonqAPI architecture. It needs to be able to execute queries on different data sources and be easily extensible to support new data sources or changing queries.

5.3 AWS Service Evaluation

The LonqAPI will be implemented as a cloud-native application on AWS as part of the Data Layer at BHS, as presented in chapter 4. This section evaluates AWS services

to fulfill the requirements of the LonqAPI architecture components described in the previous section. The first subsection discusses API services for REST, the second and third subsections evaluate services for query result and status storage, and the last subsection covers services for the query execution component.

5.3.1 REST API Service

API Gateway is a service available in the AWS cloud specifically for REST and WebSocket APIs. It offers multiple features for API management, such as API keys, Transport Layer Security (TLS) encryption, versioning, and monitoring [77]. API Gateway does not include computing resources to host the API implementation but can be integrated with other AWS services, such as Lambda, to execute the API implementation [49]. Lambda functions are on-demand computing resources to execute inside a managed or self-managed container. Their memory and execution time can be configured, but they scale automatically to handle the load of incoming requests. Lambda functions are billed per millisecond of execution time and the amount of memory configured, but their execution time is limited to 15 minutes [78], [79].

Another possibility to host web services on AWS is to use EC2 instances. EC2 offers virtual machines that can be configured with different operating systems, custom software, and hardware resources. Its advantage is that it is highly customizable and can be used for various use cases. The disadvantage is that it requires more effort to set up and maintain the virtual machines than using managed services. AWS Fargate aims to reduce this complexity by providing a serverless container platform managing the underlying infrastructure. EC2 and Fargate are priced by configured hardware resources and per second of execution time with a minimum of one minute [80], [81].

For the development of the LonqAPI, API Gateway in combination with Lambda is the best choice because it reduces operational work and offers specific features for API management. API Gateway can be used with other AWS services than Lambda, such as Fargate, but Lambda's pricing model makes it more cost-efficient for short-running infrequent requests. If the LonqAPI is used more consistently, Fargate or EC2 can be a better choice.

5.3.2 Query Result Storage Service

AWS offers multiple services to store data, such as S3, RDS, Elastic File System (EFS), and DynamoDB. However, based on the following characteristics, S3 is the best choice for the query result storage service of the LonqAPI. S3 stands for *Simple Storage Service* and is a service to store objects in a flat structure. Objects can be up to 5TB in size and are stored in buckets that can be organized in a hierarchical structure using prefixes. S3 is highly available and durable, and its pricing is based on the amount of data stored and transferred out of the service. Compared to other storage services, S3 is less expensive because no computing resources are necessary to store and retrieve data. To maintain and reduce storage, S3 offers lifecycle rules to archive or delete objects after a specified time. Objects in buckets can be encrypted, configured to be restricted to specific users or

roles, and accessed using an HTTP API. Additionally, presigned URLs can be generated to grant temporary access to objects without requiring further authentication [64]. S3 is a suitable storage service for the LonqAPI because it is not limited to a specific data source or format, can store large amounts of data for an unlimited time, and provides an easy way to retrieve it. With presigned URLs, the LonqAPI can provide query results to end-users without additional complexity for the LonqAPI implementation.

5.3.3 Query Status Storage Service

One option to store the query request status would be S3. The status can be persisted the same way as the query results as an object in a bucket. This object could be in a format like JSON and have the correlation identifier as its key to retrieve it within the request handler Lambda function. However, DynamoDB is a better choice for this use case. DynamoDB is a key-value Not only SQL (NoSQL) database service that advertises "consistent, single-digit millisecond performance, nearly unlimited throughput and storage" [82]. With on-demand capacity, DynamoDB is priced per million request units and GB of storage [83]. Like S3, DynamoDB allows the expiration of stored items after a specified time [66]. The combination of performance and low pricing for low usage makes DynamoDB a good choice for the query status storage service of the LonqAPI. It can easily be used with the correlation identifier as the key and the request status as its value.

5.3.4 Query Execution Service

As evaluated in subsection 5.3.1, a Lambda function is in charge of the client request invocation. S3 is used for the query result storage service, as explained in subsection 5.3.2. This Lambda function has to initiate a worker service to execute and process the query and store the results in S3 while the Lambda function returns the response to the client. This worker service is the query execution component of the LonqAPI architecture and should be flexible and easily extensible to support various data sources and changing queries. In the following, three different approaches are described to design this component using AWS services. This is followed by their evaluation.

Athena + Connectors

Athena is a service to analyze and query data stored in S3 and supports file formats like Comma-Separated Values (CSV), JSON, or parquet. It is used to query POM data in the Machine Data Layer at BHS, as described in section 4.2.3. The Athena API allows one to start an asynchronous queued query execution, get its status, and retrieve the results once the query is finished [84]. Results of Data Manipulation Language (DML) statements are stored by default in S3 in CSV format. However, using the Create Table As (CTAS) or UNLOAD statement, the results can be stored in other formats, such as JSON or parquet. Specialized in files in S3 with data structure defined in a Glue Data Catalog, Athena can not query data stored in other AWS services by default. For this purpose, connectors can be integrated for supported services, such as RDS, DynamoDB,

or DocumentDB. An Athena connector is a Lambda function that converts the query format and results between Athena and the specific service. For this conversion, the connector also needs a Glue Data Catalog to reference the data and its schema in the service [71].

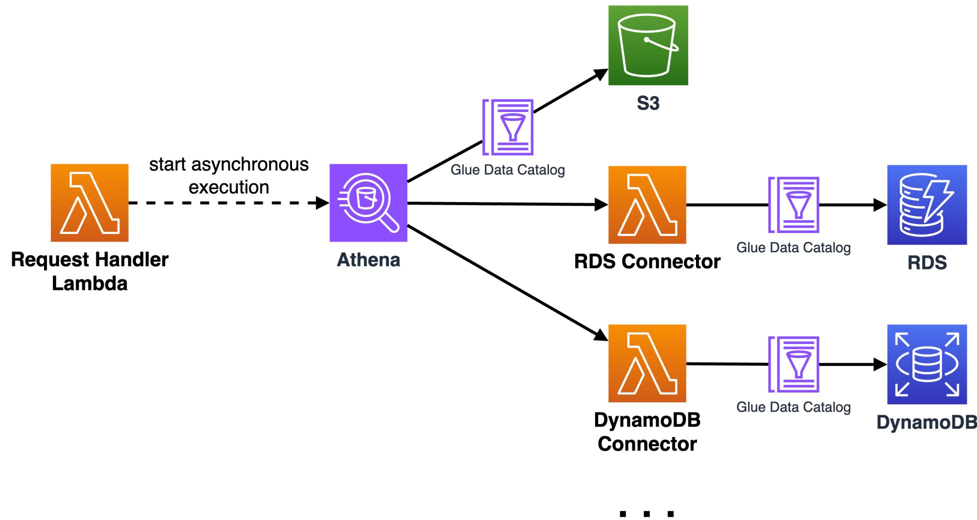

Figure 5.2: The architecture of the Athena query-handling approch in combination with Lambda connectors

Athena can be used as a query-handling approach, as shown in figure 5.2. The POM data query use case of the LonqAPI can be implemented using Athena without a connector, but each other data source than S3 would require a Lambda connector. It is relatively easy to use with standard SQL and its asynchronous query execution. If no further data processing is necessary, the query results can be stored in specific formats directly in S3 using the CTAS or UNLOAD statement. However, the need for a Lambda connector for other data sources further limits the use of Athena. AWS Lambda is limited to 15 minutes of execution time [79], which is not enough for the requirements of the LonqAPI. Another drawback is creating and maintaining a Glue Data Catalog for each data source.

Fargate Container Cluster

AWS Elastic Container Service is a fully managed service to run Docker containers on AWS. ECS offers EC2 instances and AWS Fargate as capacity options to run the containers. EC2 instances are virtual machines that can be configured with different operating systems, custom software, and hardware resources. Fargate allows less configuration and can run constantly as a service or on-demand as a task. The definition of a Fargate task specifies the Docker image to run, the amount of memory and CPU to allocate, and other configuration options. To start a task and get its current status the, ECS API with the StartTask and DescribeTasks actions can be used. A task does not have an execution limit, but in case of security patches or other maintenance, it can be stopped by AWS after a retirement notification. Both EC2 and Fargate are priced by configured hardware resources and per second of execution time with a minimum of one minute [80], [81], [85].

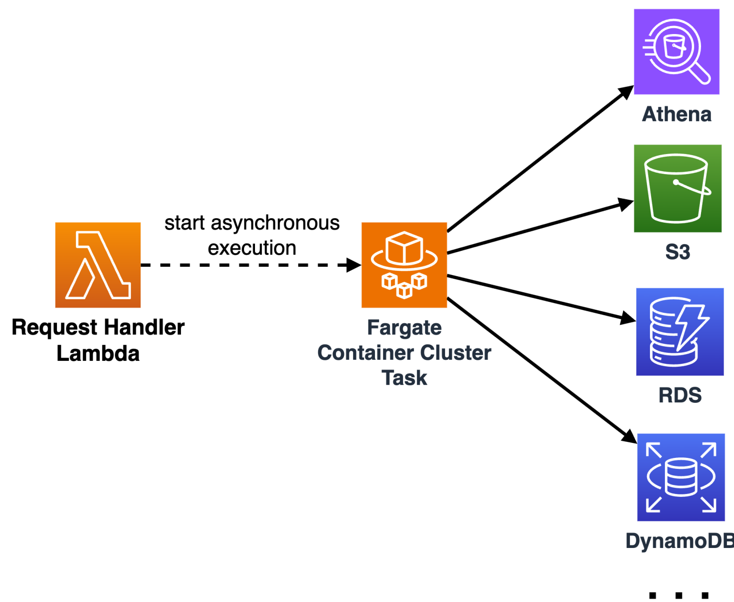

Figure 5.3: The architecture of the Fargate container cluster query-handling approach

As a query-handling approach, an ECS cluster with Fargate tasks can be used as shown in figure 5.3. The request handler Lambda function can start the Fargate task asynchronously with the query as a parameter. A new container is started for each query request and terminates once the query is finished.

Step Functions State Machine

AWS Step Functions is a service to orchestrate applications by integrating other AWS services in a workflow. This workflow is defined as a state machine consisting of a series of event-driven states. Task states can be used to perform work, such as running a Lambda function, calling another AWS service, or invoking a third-party API. Other state types allow to control the flow of the state machine, such as choice states to branch the flow based on conditions or parallel states to execute multiple tasks in parallel. A state machine task can call any AWS service by its API, but Step Functions also offers optimized integration for some services, such as Lambda, Athena, or ECS. Athena, for example, can be queried synchronously or asynchronously using the StartQueryExecution API action.

However, not all storage services can be queried directly using Step Functions tasks, e.g. RDS instances. The RDS Data Service API can only run SQL statements with a result limit of 1 MB binary response data [86]. Other solutions can be necessary to query such a service, like Fargate tasks. Another possibility to query PostgreSQL databases especially in RDS would be to prepare the database to export to S3 by installing the aws_s3 extension [65]. With this extension, the integrated ExecuteStatement task can be used by the state machine without a result limit. The input and output of a state machine task is limited to 256 KB of data. Passing larger amounts requires storing it in S3 and passing a reference to the object. Queries integrated into a state machine need to mind this data limit and should store results directly in S3, like Athena with the CTAS

or UNLOAD statement. If a task can not store results directly in S3, a ResultWriter within a Map state can write data exceeding the limit to S3, and object keys are passed to the next state.

A state machine can either be defined as a standard or express workflow. Express workflows are optimized for high-performance and high-throughput use cases but are limited to 5 minutes of execution time. Standard workflows are limited to 1 year of execution time and are only billed by the number of state transitions with 0.025 $ per 1,000 transitions. The standard workflow state machine itself does not consume any resources, only the tasks executed by it are billed. State machine executions can be started asynchronously using the StartExecution API action [87], [88].

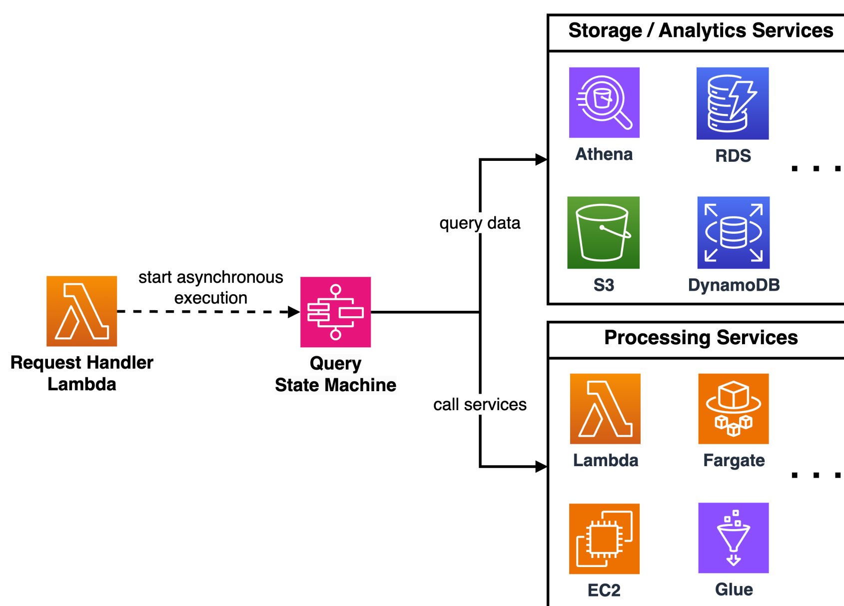

Figure 5.4: The architecture of the Step Functions state machine query-handling approach

As a query-handling approach, a Step Functions state machine can be used as shown in figure 5.4. The request handler Lambda function can start the state machine asynchronously with the query as a parameter in its input. As described in the AWS Step Functions FAQ [89], state machines can be better than queue-based approaches with AWS SQS. They offer flexibility by allowing integration with any other AWS service and features to facilitate application development. In figure 5.4, the boxes around the storage and processing services represent that they are not called by another executing service like Fargate or Lambda but as part of the state machine, depending on the query operation. This way, a state machine can consistently cover all steps from querying the data to storing the processed results in S3. Athena and other services can be called directly without other billed resources, and the state machine itself is not billed for waiting time. If a service can not write its output directly to S3, a ResultWriter can do this in a Map state. Database queries or other services that can not be called directly by a state machine task can be executed by a Lambda Function or Fargate, depending on the execution time.

Evaluation of Query-Handling Approaches

Each of the three query-handling approaches described in the previous subsections is a possible solution for handling long-running queries in AWS. The Athena approach is the easiest to implement for the POM data use case described in section 4.2.3. However, no data processing is supported after the query execution without further services. Athena is also only suitable for other data sources with a connector, which requires additional effort to implement and maintain. These connectors are limited to 15 minutes of Lambda execution time and, therefore, are unsuitable for the LonqAPI. The Fargate approach is a simple solution to execute queries and process results in self-defined containers. The drawback is that managing the container cluster with a suitable configuration requires effort to reduce costs. In comparison, the Step Functions approach does not require computing resources itself. This reduces infrastructure management effort and costs but requires more effort to implement the state machine and its tasks. Like a Fargate container, the state machine can combine all steps for query execution and result processing in a consistent way. However, if implemented correctly, the state machine can also be extended by further states to support future requirements that build on data processed by the state machine. This extensibility and the reduced costs make the Step Functions approach the best choice for the LonqAPI.

5.4 Architecture Overview

This section describes the architecture of the LonqAPI based on the architectural decisions and AWS service evaluation in the previous sections. Figure 5.5 shows an overview of the described architecture. It uses API Gateway as the REST API service, Lambda to handle the client REST requests, DynamoDB to store the request status, and S3 to store the query results. Step Functions State Machines are used to execute the queries asynchronously and process the results into a format usable by the client.

A client sends a data request via Hypertext Transfer Protocol Secure (HTTPS) to the API Gateway REST API, which invokes a Lambda function to handle the request. The request handler Lambda function parses the request and starts a Step Functions state machine asynchronously with the necessary parameters for the query as input. To return a synchronous response, the Lambda function continues its execution decoupled from the state machine, generates a correlation identifier, and saves the request status in DynamoDB. The correlation identifier is returned to the client as part of the response. The client can use this identifier to poll the query status and get presigned URLs to retrieve the query results from S3. The state machine executes the query defined by the client request and stores the results in S3. Once the query is finished and the client polls for its status, the request handler Lambda function can generate the presigned URLs, update the DynamoDB status item and return the response to the client.

There are different ways to extend this architecture further. For example, multiple or a single Lambda function can be used to handle client requests. All queries can be executed by a single state machine or separately for each query. These and other design decisions are discussed in chapter 6.

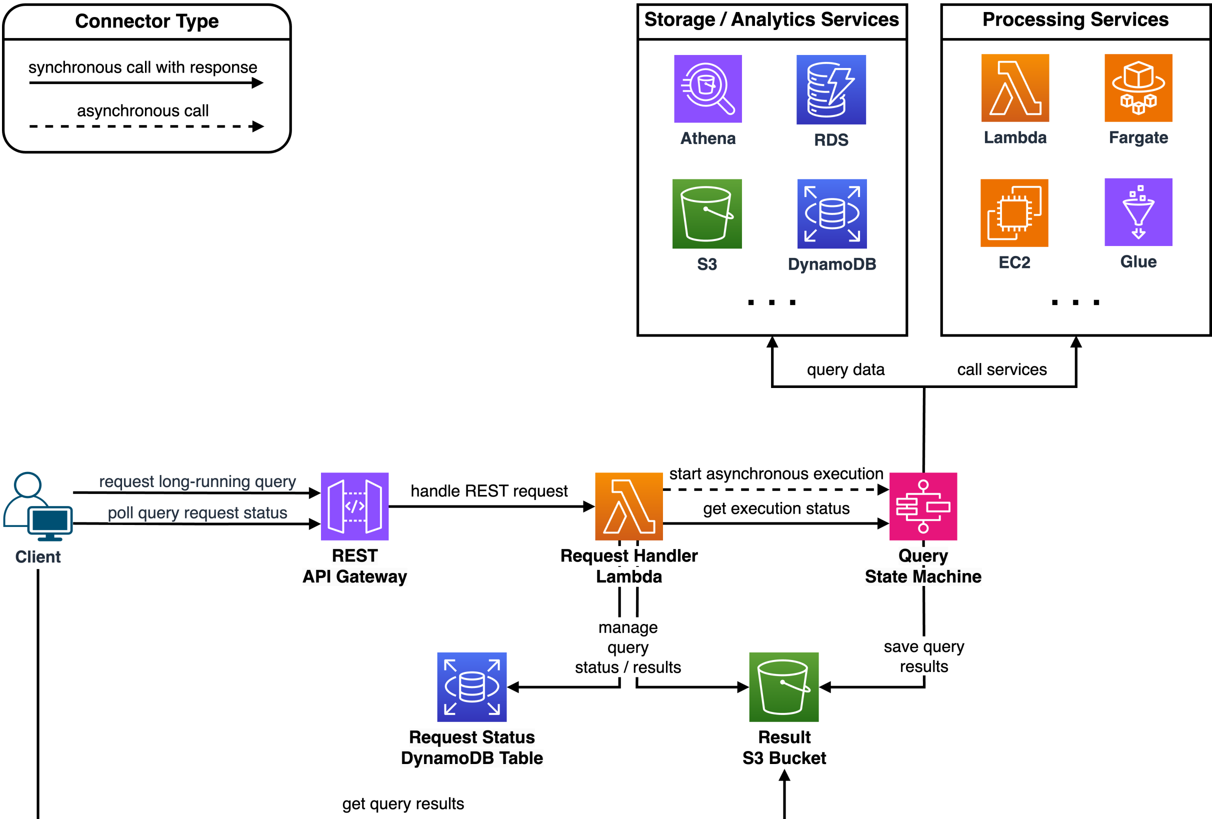

Figure 5.5: The LonqAPI architecture overview

Chapter 6

Design Decisions

As stated in section 4.3, data based on the Athena POM query has to be provided, with additional performance and implementation flexibility requirements. This data should be accessible via a REST API served by AWS API Gateway and AWS Lambda, as evaluated in chapter 5. The queries are executed within a Step Function state machine, and the results are stored in S3, accessible for the client via presigned URLs. This chapter describes the design decisions for the LonqAPI implementation in chapter 7.

First, the REST API is designed in section 6.1, including the polling pattern with general design practices for necessary endpoints. Then, the handling of API Gateway events with AWS Lambda is evaluated in section 6.2. The third section describes decisions concerning query handling within state machines, followed by a summary.

6.1 REST API Design

This section makes decisions on the design of the REST API. Existing approaches are described to implement the polling pattern in the LonqAPI for data based on long-running queries. Furthermore, general RESTful practices are considered to design the necessary endpoints for the Athena POM query.

6.1.1 REST Polling Pattern

Based on the HTTP protocol, REST is a synchronous request-response communication architecture. To enable asynchronous communication, the operation needs to be divided into two or more synchronous requests. To achieve this for the LonqAPI, the polling pattern is used. It is described in the following paragraph based on designs presented by Richardson and Ruby [90, pp. 228-230] and Allamaraju [91, pp. 19-22].

To start and track a long-running operation, the server provides a resource representing the operation. The operation can also be called a task, job, or, in the case of the LonqAPI, a long-running query. To create a new long-running query, the client sends a POST request to the resource with all associated parameters. Even if the client wants to retrieve

data from the API, this should not be a GET request because the purpose is to create a new resource. The server initiates the query and returns a response with the HTTP status code 202 (Accepted), a Location header with the URI of the created resource, and a response body with the current resource representation. The resource representation contains the status of the query and other related information. The response can also include a Retry-After header to indicate when the server estimates the operation to be completed [92, p. 300]. The URI of the resource can be used to poll the status of the query by sending GET requests. To polling requests the server responds with the query representation and a status code that can depend on the current execution status. Allamaraju [91, pp. 19-22] uses a 303 (See Other) status code with a Location header pointing to the result once the query is completed and a 200 (OK) status code with the query representation otherwise.

For the LonqAPI, the long-running query resource is defined. It can be created with a POST request that returns the resource URI in the Location header, a query representation in the body, and the status code 202 (Accepted). Using this URI, the client can poll the query status with GET requests. It contains a unique ID as the correlation identifier, which is also included in the query URI. Since the query can result in multiple files, the resource contains a list of presigned S3 Uniform Resource Locators (URLs) to download the data by sending GET requests. The Location header can only point to a single other resource [93, p. 134], so it is not used to point to the results. Instead, on completion of the query, its resource is updated with the result URLs, and the status code 200 (OK) is returned.

6.1.2 RESTful API Design Practices

The following paragraphs summarize design practices presented by Richardson and Ruby [90, Ch. 5] and De [43, Ch. 3]. These practices are limited to the use case of the LonqAPI and do not claim to be complete.

Hypermedia as the Engine of Application State (HATEOAS): Resource representations can include links to related resources. This allows clients to navigate dynamically through the API without knowing the URI structure.

URI design: The resource name should be a noun, not a verb. Domains and subdomains should be used to separate and organize resources in a hierarchy. A collection resource can group multiple similar resources, should have a separate URI, and be named in the plural. Non-hierarchical elements can be separated by a comma (,) or semicolon (;). Hyphens (-) or underscores (_) can be used to separate words in a resource name. URIs are case-sensitive, but lowercase is preferred.

Request variable placement: The query parameters can be placed in the path, query string, body, or header of the request. The path should be used for variables that are part of the resource identification or hierarchy. The query string should be used for variables that are input to an algorithm. The body should be used for variables that are part of a representation but can also be used for more complex inputs that do not fit in the query string. The header should be used for variables not part of the resource or

representation but for authentication or other purposes.

Response codes: The response code indicates the result of the REST request. Section 6.1.1 describes the response code 202 (Accepted), indicating that an asynchronous operation is initiated. The response code 200 (OK) indicates that the request was successful, as well as for polling requests that return the current status of the operation, even if it is not completed or failed. If a client requests a resource that does not exist, the response code 404 (Not Found) is used. The response code 400 (Bad Request) should be returned for invalid requests for which the client is responsible. If the server is responsible for the error, the response code 500 (Internal Server Error) should be returned.

Response body: The response body should contain the requested resource representation. The representation can be in different formats, but JSON is recommended for serialized data structures.

6.1.3 LonqAPI REST Endpoints

As described in section 6.1.1, a long-running query is represented by a resource and can be created by a POST request and polled by a GET request. In the context of this thesis, only the Athena POM query described in section 4.2.3 must be implemented. The following paragraphs present the LonqAPI REST endpoints based on the practices described in the previous section (6.1.2) and requirements from section 4.3.

Create POM Query Endpoint

The LonqAPI consists of two endpoints, one to create a POM data query and one to poll the status for a long-running query. The relative path without the server address of the POM data query endpoint looks like this:

```
POST /daas-history/long-running-queries/{client_id}/process-data/
    areas/{area_id}/points-of-measurement-query
```

As defined in the requirements, *DaaS* and *history* are used apart from the API domain name in first-level path elements. The term *long-running-queries* separates long-running queries from other resources. After that, data-specific path elements are integrated to define the query to create. To identify the client that initiated and, therefore, owns the query, the *client_id* is included in the path. The *process-data* sub-path separates other data origins. An area represents a client's production facility, which is also included in the path hierarchy with the *area_id*. Because a client can have multiple areas, the *areas* sub-path is used before the *area_id* to specify a single area out of multiple possible areas. Finally, the *points-of-measurement-query* path element names the concrete query to create. It is based on process data from a specific area of a client.

This path has to be extended with the following query string parameters that are input to the query algorithm:

- **pomIds:** The IDs of the points of measurement to query as comma-separated string.

- **startTime:** The query interval start time as milliseconds since the Unix epoch [72, Sec. 4.16].
- **endTime:** The query interval end time as milliseconds since the Unix epoch.
- **imperial:** A boolean value to indicate if the query result should be in imperial units.

The query string parameters are written in camel case to be consistent with other BHS APIs, even so, lowercase should be preferred. The ID and timestamp formats are as defined in the requirements in section 4.3. The imperial parameter is optional and defaults to false, as specified in user story 1. As mentioned in section 4.2.3, the data transformation triggered by this parameter could be implemented in the client application. However, due to legacy applications at BHS, it is implemented in the LonqAPI. Since only metric or imperial units are supported, the parameter is a boolean value.

The response of this endpoint specifies the created query resource in the Location header and a 202 (Accepted) status code. The response body contains the query resource representation with the current status and other related information as described in section 6.1.3.

Long-running Query Resource Endpoint

Queries and their status can be polled with a GET request to the query resource endpoint. It is not specific to a query type and, therefore, is used for all created long-running queries. This reduces the number of endpoints and simplifies future extensions. Therefore, the path contains only variables that are relevant part of the identification of the query and not the query type. Its relative path looks like this:

```
GET /daas-history/long-running-queries/{client_id}/{query_id}
```

The *client_id* is included to identify the query initiator, and the *query_id* specifies the query to poll. The client ID is an essential part of the query resource as its owner identifier. The area ID is specific to the POM data query but may not be relevant for other queries. Therefore, it is not included in the query resource and its path.

Long-Running Query Resource Representation

The response body for both endpoints is a JSON object representing the query resource. Like its path, the query resource is independent of the concrete query type and is used for all long-running queries. It contains only the information that are general for all queries. Aligned with the Google JSON style guide [94], property names are in camel case. The query resource representation contains the following properties:

- **clientId:** The ID of the client that initiated the query as UUID version 4 [75] string.
- **queryId:** The ID of the query as UUID version 4 string.
- **status:** The query status as a string with the values *RUNNING*, *COMPLETED*, *COMPLETED_NO_DATA*, or *FAILED*.
- **resultUrls:** A list of presigned S3 URLs to download the query result with temporary access.

- **creationTime:** The query creation timestamp in ISO 8601 format [76].
- **expirationTime:** The query expiration timestamp in ISO 8601 format.
- **createQueryUrl:** The URL, including the parameters, used to create the query.
- **queryUrl:** The URL of the query resource.

The client ID and query ID are included because they are part of the query resource identification. The status is necessary for the client to poll until the query is finished. It is set to *RUNNING* when the query is created. *COMPLETED* and *FAILED* are sufficient as final statuses. However, the *COMPLETED_NO_DATA* status is used to avoid confusion with the *COMPLETED* status when no data is available for the query. Other status definitions can be added in the future. The creation and expiration timestamps are included to indicate the query lifetime. The query ID is sufficient to correlate the created query with the original request. However, the *createQueryUrl* is appended to the query resource representation to simplify the API usage for humans by providing the URL used to create the query. The query URL is included to simplify the polling process for the client. The list of result URLs is empty when the query is created and updated once results are available. Listing A.3 in the appendix shows an example query resource representation.

6.2 AWS Lambda Request Handling

This section makes decisions on how to handle REST API requests with AWS API Gateway and Lambda. It evaluates the advantages and disadvantages of a monolithic and single-purpose Lambda function approach. Furthermore, it compares Python frameworks compatible with the Lambda runtime to handle API Gateway events.

6.2.1 Monolithic vs. single-purpose Lambda Function

API Gateway can trigger a Lambda function with a REST API event. This event contains information like the HTTP method, path, query string parameters, and headers. One Lambda function can be used for multiple endpoints, grouped by the path and HTTP method. This allows different Lambda function architecture designs to handle API Gateway events. One extreme is to use a single-purpose Lambda function for each endpoint. The other extreme is to use one monolithic Lambda function for all endpoints. A mix of both is also possible with micro functions for groups of endpoints. Each approach has advantages and disadvantages for infrastructure implementation, deployment, maintenance, and performance. These are discussed in various blog posts like [95] and [96] and mentioned in AWS documentation and guides [97], [98]. Table 6.1 summarizes the advantages and disadvantages of the monolithic and single-purpose Lambda function approaches.

Multiple functions can be configured and deployed independently, which gives more granularity over Identity and Access Management (IAM) permissions, resource usage, and other configurations. Package size can be reduced because it only contains the code needed for the endpoint, and the deployment time of the single function can be faster. The complexity per function is reduced because its scope is smaller. Tracing and

Pros	Cons
Single-Purpose Lambda Function	
Granular permissions and resources Reduced package size per function Lower complexity per function Separate tracing and logging Independent scaling Independent testing Better reusability	Growing infrastructure complexity Harder code sharing Deployment of multiple functions Difficulty in permission maintenance More frequent cold starts Harder runtime switching
Monolithic Lambda Function	
Easier maintenance and extension Better code sharing Lower infrastructure complexity Deployment of only one function Less frequent cold starts Framework usage benefits Easier runtime switching	Function can become complex Less granular permissions Larger package size per function Bigger deployment impact Low reusability Joined tracing and logging

Table 6.1: Comparison of monolithic and single-purpose Lambda functions for API Gateway request handling

logging activities are better separated and facilitate debugging. Multiple functions can benefit from the Lambda scaling model and can be scaled independently, depending on the specific usage. If the functions perform generic tasks, they can be reused in other projects. Depending on the implementation, changing and testing a small function can be easier because it has fewer dependencies and is limited to a specific task. However, the number of functions can grow quickly, and the whole application can become complex if the functions are poorly organized. To benefit from small package sizes and fast deployment times, the functions should be independent and not share dependencies. This can lead to code duplication and maintenance overhead. If multiple functions share dependencies, the deployment time can increase because the changes must be deployed to all dependent functions. To achieve more granular permissions, the functions need to be configured with different roles and policies, which can be difficult to maintain, and in the case of IaC tools like AWS CDK, infrastructure code increases. The number of functions can also affect performance because the Lambda runtime needs to be initialized for each function, causing cold start delays more frequently.

A monolithic function can be easier to maintain and extend because all code is in one place. Code and dependencies can be shared between endpoint routes, and code duplication is avoided. The infrastructure code is simpler, and deployment time can be reduced because only one function needs to be configured and deployed. Cold starts can be reduced because the Lambda runtime is initialized only once and used for multiple purposes. With AWS API Gateway events that trigger the function, it is possible to use a framework to resolve the requests, avoid boilerplate code, and extend the implementation with additional functionality. A REST framework can also be used for multiple smaller functions, but this makes them share dependencies, and changes

can affect multiple functions and their deployment. A single function can make it easier to switch from Lambda to another runtime like AWS Fargate or EC2. However, the function can become complex and difficult to understand due to its larger scope. IAM permissions are less granular, and a execution can have more permissions than needed for a specific endpoint. The package size is bigger, and the single deployment and cold start time can be longer. Changes to one endpoint affect the whole application and need to be considered. The broad usage makes it hard to reuse the function in other projects. Logging and tracing activities are not separated and can be harder to debug.

The LonqAPI REST API is designed with a monolithic Lambda function handling API Gateway events for all endpoints. The main reason is to ensure flexibility and extensibility without increasing infrastructure complexity and deployment time. The API can be extended with additional endpoints and functionality without deploying additional functions. This avoids increasing infrastructure and deployment time, especially if the API can grow to a large number of endpoints. If the API requirements change, the implementation can be moved to another runtime like AWS Fargate or EC2 more easily. A framework is used to reduce boilerplate code, add functionality, facilitate debugging, and improve the developer experience. The framework is evaluated in the following section (6.2.2). In the case of critical permissions or runtime requirements, the function can still be split into multiple with different policies and configurations.

6.2.2 REST API Python Framework

The REST API is realized with AWS API Gateway and a single Lambda function to handle all endpoint requests. As defined in the requirements in section 4.3.2, the Lambda function is implemented in Python. The API Gateway event is passed to the Lambda as input to the function handler and contains information like the HTTP method, path, query string parameters, and headers. The Lambda function needs to parse the event, resolve the request, perform the necessary actions, and return a response. To reduce boilerplate code, add functionality, and improve the developer experience, the implementation is based on a framework. This section evaluates different Python frameworks for REST APIs and their compatibility with AWS Lambda and API Gateway events.

Framework Comparison

To be applicable to the LonqAPI, the REST API framework must meet specific requirements. These minimal requirements for a framework are:

- Python 3.10 compatibility
- Resolve requests from API Gateway in the Lambda function runtime
- Parse request parameters in the path, query string, and body
- Return a response with a defined status code, header variables, and body

A framework that meets these requirements can be used to implement the LonqAPI REST API. The idea of using such a framework in an AWS Lambda function implementation with API Gateway events can be seen in listing A.4 in the appendix.

To compare and evaluate applicable frameworks, further criteria are defined to consider the framework benefits and drawbacks for the LonqAPI use case. Included is also the feature of an automatic OpenAPI specification [99] generation for the REST API documentation, which some frameworks offer. This standard can be used to define and document the API and satisfy the LonqAPI requirements in user story 4 in section 4.3. Performance is not considered because of a lack of comparable benchmarks for the use case of this thesis. There are benchmarks comparing most of these frameworks, like the independent TechEmpower benchmarks [100]. However, their quality is not scientifically guaranteed, and necessary adapters for AWS Lambda and API Gateway events are not considered. Also not considered is testing utilities and documentation quality because all frameworks offer both in a sufficient way. However, the simplicity criteria takes into account the quality of the documentation. The following criteria are defined to evaluate the frameworks:

- **Features:** The framework offers rich features for the LonqAPI use case.
- **Community Support:** The framework is used by a big community, has a large number of contributors, and is actively developed and maintained.
- **Simplicity:** The framework is easy to learn and use.
- **OpenAPI:** The framework can automatically generate an OpenAPI specification for the REST API documentation and provide it in a standard file format like JSON or YAML Ain't Markup Language (YAML).
- **Adapter:** The AWS Lambda and API Gateway events adapter is actively developed and maintained.

The following paragraphs present applicable frameworks that are evaluated based on the criteria. For simplicity, only the frameworks and adapters for AWS Lambda and API Gateway events are evaluated, not further extensions. To measure the community support, the GitHub repositories of the frameworks are analyzed with their number of stars, contributors, and merged pull requests.

AWS Lambda Powertools: AWS provides the Lambda Powertools library [97] that contains utilities for Lambda functions and includes a resolver for API Gateway REST APIs. This *APIGatewayRestResolver* fulfills the minimal requirements for the LonqAPI and does not need a adapter like the other frameworks. Its features are limited, but it enables request and response validation and OpenAPI specification generation. This OpenAPI specification is only available by an optional Swagger UI [101] endpoint and not in file formats like JSON or YAML. Therefore, it is relatively simple to use, and the documentation is good with multiple examples. Its software is actively developed and maintained by, Amazon with 1059 merged pull requests for the year 2023 at the time of writing [102]. However, the usage is limited to AWS Lambda and API Gateway, and so is its community, with 2.5 k stars and 130 contributors. It is worth mentioning that this library includes more utilities for Lambda functions, and the REST API resolver is not its primary purpose like for the other frameworks.

Flask with awsgi: Flask [103] is a popular Python microframework with core functionality. It is a Web Server Gateway Interface (WSGI) framework for web applications and cannot be used with API Gateway events by default. However, it can be used

with the awsgi adapter [104] to run on AWS Lambda and resolve the events. With its small core, Flask is built to be simple and easy to extend during development. Flask without extensions does not offer further features like request validation or OpenAPI specification generation. On GitHub, it has over 65 k stars and 713 contributors, but the necessary awsgi adapter repository has only 197 stars and 14 contributors. Flask is still actively developed, but compared to AWS Lambda Powertools, it has fewer merged pull requests, with 111 for 2023.

Django with Mangum: Django [105] is like Flask a popular WSGI framework with Asynchronous Server Gateway Interface (ASGI) support since version 3.0. It is a full-stack high-level framework with many features for more purposes than Flask. To make it compatible with AWS Lambda and API Gateway events, the awsgi or the Mangum [106] adapter can be used. Mangum is a wrapper for ASGI applications and should be preferred over awsgi because of its higher popularity, with 1.5 k stars and 28 contributors. Django has the largest community of the evaluated frameworks, measured by 74 k stars and more than 2.4 k contributors. With 731 merged pull requests in 2023, it is also actively maintained. However, its many features and use cases make it more complex than the other frameworks, and it requires more time to learn. Django does not offer native OpenAPI specification generation.

FastAPI with Mangum: FastAPI [107] is an ASGI framework based on Starlette [108] and Pydantic [109]. FastAPI is a newer framework that aims to be “high performance, easy to learn, fast to code, ready for production” [110]. With Pydantic, it integrates type checking and validation for requests and responses. Further usable features are controller classes, dependency injection, and native OpenAPI specification generation. To run on AWS Lambda, the Mangum adapter can be used like for Django. FastAPI is relatively new but already has a large community with 66 k stars and 542 contributors. Although it is actively developed with 329 merged pull requests in 2023, FastAPI is mainly developed and maintained by one person, and some issues are open for a long time, with the oldest open issues dating back to 2019.

Litestar with Mangum: Litestar [111] is a new ASGI framework like FastAPI based on Starlette and integrates Pydantic for type checking. Formerly named Starlite, it was renamed Litestar to avoid confusion with Starlette, whose dependency was removed in November 2022 after growing more independent. Litestar already has similar features like FastAPI and some more, like caching and server-side session support. It is relatively new and has a smaller community with 3.7 k stars and 170 contributors, but compared to FastAPI, it is maintained by three chosen maintainers and represented by an organization. This more independent development is also reflected in the number of merged pull requests, which is over 1000 for 2023. With Mangum, it can be used on AWS Lambda and API Gateway events like FastAPI or Django. It has good, detailed, structured documentation with multiple code examples explaining its relatively easy usage. Like FastAPI, Litestar has native OpenAPI specification generation that can be accessed via endpoints as JSON or YAML files or as Swagger UI.

Framework Evaluation

The described frameworks are evaluated based on the criteria from section 6.2.2. To compare the frameworks, each criterion is assigned a rating represented by a rating symbol in table 6.2. The rating symbols are defined in table 6.2.

Rating Symbol	Meaning
+ +	The criteria fully applies.
+	The criteria mostly applies.
o	The criteria partly applies.
-	The criteria minimally applies.
- -	The criteria does not apply at all.
?	The criteria is not applicable or not evaluated.

Table 6.2: Framework rating symbols

Framework	Features	Community Support	Simplicity	OpenAPI	Adapter
AWS Lambda Powertools	o	o	+ +	o	?
Flask with awsgi	-	+	+	- -	-
Django with Mangum	+ +	+ +	-	- -	+
FastAPI with Mangum	+	+	+	+ +	+
Litestar with Mangum	+	+	+	+ +	+

Table 6.3: REST API Python framework comparison

Litestar with Mangum is chosen as the framework for the LonqAPI REST API. This decision is due to the combination of simplicity and useful features, especially the native OpenAPI specification generation. FastAPI offers similar characteristics but is mainly developed, maintained and represented by one person, which can be a risk for the future. Litestar is less popular, but it is based on an organization, has more maintainers, and is already in a stable version, which makes it more reliable.

6.3 AWS State Machine Query Handling

This section describes the design decisions for the query execution and result handling using AWS Step Functions state machines.

6.3.1 State Machine Design

AWS Step Functions state machines are used to execute the long-running queries, transform the results if necessary, and store them in S3. The service, in general, is not specific to a particular use case and can be used for different purposes. Therefore, the following paragraphs describe the design decisions for the state machines in the context of the LonqAPI.

Monolithic vs. single-purpose State Machines

As for the REST API request handling with AWS Lambda, the query execution can be designed with a monolithic state machine or multiple state machines. The term

monolithic state machine is not defined by AWS or other sources and refers in this thesis to an AWS Step Functions state machine that executes all queries to all data sources within the LonqAPI architecture.

The advantages of a monolithic state machine are the reduced infrastructure complexity and deployment time. The state machine can be extended with additional states and functionality without deploying additional infrastructure. Advantages for single purpose state machines are the reduced complexity per state machine and the possibility to define concrete steps with minimal IAM permissions without the need to consider higher flexibility. However, the number of state machines, infrastructure code and complexity, and deployment time can increase quickly. Therefor, the approach of multiple generic state machines as a middle ground between monolithic and single-purpose state machines is chosen. This way, the number of state machines is reduced, but permissions and complexity are separated between the state machines.

Generic LonqAPI State Machine

A state machine for the LonqAPI is designed to be a generic interface to a single data source without specific query logic. The data source can be a database, another API, or, like in the case of the LonqAPI, Athena, with permissions to read specific S3 buckets. The state machine offers input parameters to define the specific query and transformation logic. This input is passed in as required by the service in JSON format [87], and so is the output of each state. Large data cannot be passed between states, so it needs to be stored in S3, and states pass references to the objects. These references are passed consistently in the output of a state as a JSON object with the keys *ResultBucket* and *ResultKeys*. A query state like an Athena query returns different outputs and needs to be transformed to fit the consistent output. However, this is not possible for some queries without further overhead, so the query output is returned in such cases to avoid increased complexity and costs by billed computing resources.

Athena POM Query State Machine

To integrate the Athena POM query described in section 4.2.3, a single state machine is designed to run Athena queries on the S3 bucket containing the data. It has a query state running the *StartQueryExecution* Athena API call as a synchronous job. The concrete query is not defined in the state machine but passed as *QueryString* and *QueryParameter* inputs. For the query, the UNLOAD command is used to store the results directly in S3 to reduce runtime costs. This command does not return the object keys of the result directly but with an S3 location containing a CSV file that includes these keys. An example response to such a query is shown in listing A.5 in the appendix. As mentioned in the previous paragraph, instead of a consistent output with the result keys, the query output is returned in this case because the query command specifies the result location in S3, and the Lambda function defining this query can list the result files directly.

6.3.2 Data Transformation

Data retrieved from a data source can be transformed within the state machine. This represents the Transformation step in the ETL process described in section 3.2.2. To transform the data, a Lambda function is used as a state in the Athena POM query state machine. A Lambda function is limited to 15 minutes of runtime and a maximum of 10240 MB memory [79]. This is enough for transforming the data of an Athena POM query in chunks. For other data sources that need more time or memory, a different service like AWS Fargate can be used. The input of such a transformation state includes the output of the previous state, but depending on the state machine definition, it can also include the original state machine input.

In the case of the Athena POM query, the data needs to be transformed to imperial units if requested, as defined in user story 1 in section 4.3. Therefor, the transformation function is integrated as an optional state only executed if the *imperial* query string parameter is set to *true*. The Lambda then loads the data from S3 in chunks, transforms it, and stores it in different S3 locations. The transformation uses the existing utility function in the *mdl_data_access* module. The output of the transformation state is the consistent result output with the result bucket and keys mentioned in section 6.3.1.

6.3.3 Result Data

The data retrieved from a query within a state machine is stored in S3 and passed to the next state, as described in section 6.3.1. Each query or transformation state stores the data in a different S3 location. This fulfills the purpose of a staging area in an ETL process, as mentioned in section 3.2.2. To reduce the number of S3 buckets and keep the infrastructure simple, the system uses a single bucket for all result data and separates the data with prefixes.

The Athena UNLOAD command supports different file formats like JSON, text, and parquet. The parquet format is a columnar storage format optimized for performance and compression [70]. Data analysis tools like Apache Spark [112] or libraries like Pandas [73] can read parquet files directly. JSON is a human-readable format that is a good choice for REST API response bodies, as mentioned in section 6.1.2. However, compared to parquet, it is not optimized for performance and compression of large data [113], so the parquet format is chosen as the default format for the query results.

6.4 Summary

This chapter presented design decisions for the LonqAPI architecture and implementation. Two REST API endpoints, their parameters and a uniform resource representation are defined to create and poll long-running queries. Requests are handled with a monolithic Lambda function using the Litestar framework. The query execution is designed with a generic Step Functions state machine which stores the results in a single S3 bucket. Optional result processing is integrated as a Lambda function execution state.

Chapter 7

Implementation

This chapter describes the implementation of the LonqAPI prototype, which realizes the architecture described in chapter 5 and the design decisions in chapter 6. First, the overall project structure is described, followed by the implementation of the *mdl-lonq* library as one of the subprojects. The integration process of new long-running queries is defined based on this library. As an example, the Athena POM query and its result processing is presented. Results of the implementation process are integrated into the *mdl-daas-history* subproject, described in the following sections. The chapter ends with the implementation of a general session management utility in Lambda functions used by the LonqAPI prototype and a summary.

7.1 LonqAPI Monorepo Subprojects

The request dispatching in an AWS Lambda function and the query execution in a Step Functions state machine are decoupled within the LonqAPI architecture described in chapter 5. In the implementation, this decoupling is reflected by separating the REST API and the state machine infrastructure in two subprojects in the monorepo, *mdl-lonq* and *mdl-daas-history*. To follow the Data Layer convention, the project names start with *mdl-*. The following paragraphs describe the two subprojects and their usage. Figure 7.1 shows a simplified package structure of the LonqAPI implementation with the two subprojects as their base packages.

7.1.1 mdl-lonq Subproject

The *mdl-lonq* subproject serves as a library for other applications, including *mdl-daas-history*. It contains all necessary infrastructure and logic code to help implement AWS Step Functions state machines handling long-running queries. This includes a generic state machine definition and Lambda function implementation to transform, process, and store query results in an S3 bucket. The library aims to facilitate all steps in the decoupled query execution process within a state machine, independent of the concrete usage scenario, but keeping the flexibility to extend and customize the implementation. It

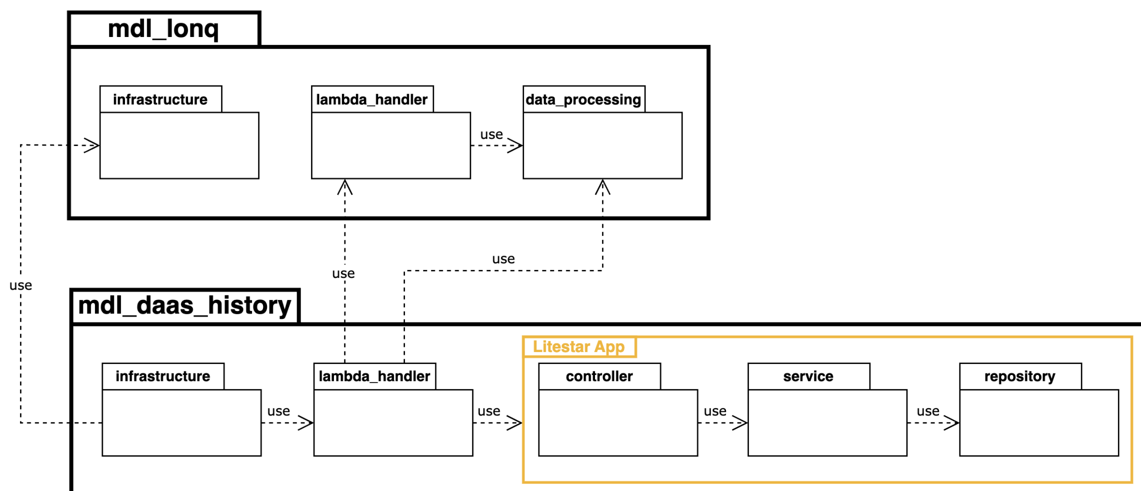

Figure 7.1: The simplified LonqAPI implementation package structure

does not instantiate any infrastructure as AWS CDK stacks itself but provides constructs to do so. It contains neither business logic nor application-specific code like API-specific functionality like the query status storage in DynamoDB.

The goals of the *mdl-lonq* subproject library are:

- Enable but do not specify queries on data sources
- Enable chunked processing of results
- Enable but do not specify transformation steps
- Provide result data in source-agnostic format
- Keep flexibility to create new state machines with different data sources
- Keep flexibility to integrate new states for future functionality
- Allow easy and intuitive state machine instantiation
- Avoid code duplication

7.1.2 mdl-daas-history Subproject

The *mdl-daas-history* subproject implements the LonqAPI REST API as the prototype of the DaaS history API in the BHS Machine Data Layer. The project defines and instantiates infrastructure and includes request-handling logic in Lambda functions. It uses the Litestar REST API framework, as evaluated in section 6.2.2, to implement the REST API endpoints and the *mdl-lonq* library to define custom state machines and result-processing Lambda functions within a CDK stack. For consistency reasons, this CDK stack is located outside the *mdl-daas-history* project folder in a separate infrastructure monorepo subproject for the whole Data Layer called *mdl-cdk-core*. In the context of this thesis, the infrastructure stack is described as part of the *mdl-daas-history* project, ignoring the fact that it is not implemented in the same project folder.

7.2 mdl-lonq Library Implementation

This section outlines the implementation of the *mdl-lonq* library introduced in section 7.1.1. The library contains three main packages described in the following paragraphs with their most important classes and functions.

7.2.1 State Machine CDK Constructs

One purpose of the *mdl-lonq* library is to facilitate the definition of state machines to execute and process long-running queries. Therefore, it provides CDK constructs that define generic and extendable state machines. A state machine can be instantiated using the CDK *StateMachine* class. To define its states, the *StateMachine* class requires a *DefinitionBody* object that contains the state machine structure. This definition can be created from a *Chainable* object that represents the first state of a *Chain* of states. Such a *Chain* can be extended by adding another *Chain* or a *Chainable* object. Each *State* class implements the *IChainable* interface and can be used to start a *Chain* or extend an existing one. A *State* can be a *Task* to call an AWS service or a third-party API. Other *State* types, like *Choice* or *Parallel*, can control the state machine flow [114].

The *mdl-lonq* library provides *Construct* classes that use the chaining concept to define state machines for long-running queries. The *BaseLonqStateMachine* is an abstract class that defines the idea of a long-running query state machine. It separates the state machine into three sections of the query handling process represented as *Chain* objects: the pre-query chain, the query chain, and the post-query chain. The class implements the Template Method design pattern [115, pp. 325-330], [116, pp. 257-270] to define the state machine structure and allows subclasses to customize it by overriding its methods. These methods include the creation of each *Chain* object for the three sections of the state machine and one to set permissions for its execution role. Only the query chain method is abstract and must be implemented by subclasses. This aims to facilitate the definition of a state machine by bundling case-independent structure and functionality and providing only necessary customization. Although the template method that defines the structure is called *build*, this class does not follow the Builder design pattern [115, pp. 97-106], [116, pp. 75-90] because no director class is used to construct the state machine. However, the implementation is inspired by its idea to "separate the construction of a complex object from its representation" [115, p. 97] to facilitate the creation of state machines. The *BaseLonqStateMachine* class expects a *BaseLonqStateMachineConfig* object as a constructor parameter that contains all necessary configuration parameters for the state machine construction. This includes a S3 *IBucket* object and prefix string to define the S3 bucket where query results are stored. The state machine has its own storage location in S3 and can store results in multiple subfolders. The *BaseLonqStateMachine* class also defines the S3 prefix for raw query results and grants appropriate read and write permissions to the state machine execution role. This construct and its configuration class are shown as a diagram in figure 7.2 and documented in listing A.6 in the appendix.

The *BaseLonqStateMachine* class is abstract and must be subclassed to implement the query chain and optionally override the other chain methods. The *ProcessedLonqStateMa-*

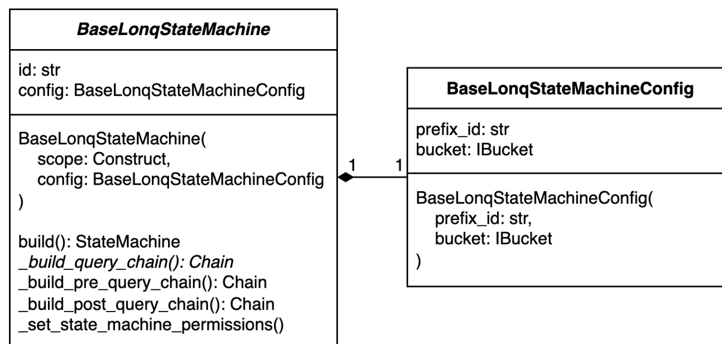

Figure 7.2: The abstract *BaseLonqStateMachine* construct and its configuration class

chine class is another abstract subclass of *BaseLonqStateMachine* that implements the post-query chain to process query results in a Lambda function optionally. This Lambda function can be passed in the configuration object to the constructor and is only executed if the state machine execution input contains a *Processor* key. To store the processed results in a separate S3 prefix, permissions are granted to the lambda function execution role to write to this S3 location.

7.2.2 Generic Query Processor Lambda Handler

Another purpose of the *mdl-lonq* library is to facilitate the processing of query results. Therefore, it provides a generic Lambda function handler that can be configured to process query results in a state machine. The idea of this handler is to provide only the data-specific functionality as a configuration object and keep the rest of the implementation generic. This is inspired by other library implementations like the AWS Lambda Powertools APIGatewayRestResolver [102] or the Mangum ASGI adapter [106]. Both libraries offer a class that can be instantiated with configuration parameters or a complex object. The class instance can then be called as the Lambda handler and executes the application logic configured in the constructor. This concept is not used within a class but is implemented in the *create_handler* function, simplified and documented in listing A.8 in the appendix, which returns a lambda handler function. As a configuration parameter, this function expects a dictionary that defines how to process the query results based on the lambda event. This input dictionary has the processor name as the key and a *Callable* as the value to create the processor from the lambda query event. This dictionary itself has similarities to the *Factory Method* pattern [115, pp. 107-116], [116, pp. 68-70] by returning a value based on a key, and the *Callable* value is inspired by the *Abstract Factory* pattern [115, pp. 87-95], [116, pp. 70-72] to dynamically create the processor object without knowing its concrete type. The *Callable* returns an object that implements the abstract *QueryProcessor* class defined in the *data_processing* package. Therefore, the lambda event must contain state machine context information like the S3 result location and query inputs like the query ID and query state output.

The *QueryProcessor* class is an abstract class that defines the interface for query result processors. It expects a *LonqExtractionIterator* object as a constructor parameter that provides the query results in chunks. Within the template method [115, pp. 325-

330], [116, pp. 257-270] *process*, the *QueryProcessor* class iterates over the query results and calls the methods *_transform_element* and *_persist_element* for each element. Both methods can be overridden by subclasses to customize the processing. The *LonqExtractionIterator* class implements the *Iterator* pattern [115, pp. 257-271], [116, pp. 203-207] using Python's internal *__iter__* and *__next__* functions. It is also abstract and must be subclassed to be instantiated.

In the package structure shown in figure 7.1, the Lambda handler creator function is represented as the *lambda_handler* package, and query processors and iterators are located in the *data_processing* package.

7.3 Long-running Query Implementation Process

The *mdl-lonq* library is designed to facilitate the implementation of long-running queries in AWS Step Functions state machines. This section describes how the library can be used to implement a new long-running query by following a process. The first subsection describes the process in general, and the second subsection describes the implementation of the Athena POM query state machine as an example.

7.3.1 Process Definition

A process is defined to implement a new long-running query using the *mdl-lonq* library constructs and utilities. This process is documented in the flowchart figure A.2 in the appendix and described in the following paragraphs. A simplified version of the process, without branches, is shown in figure 7.3. Implementation results of this process can extend the *mdl-lonq* library or be implemented in a separate project like the *mdl-daas-history* subproject.

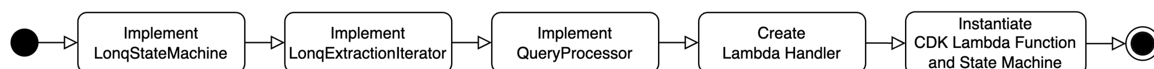

Figure 7.3: The simplified process of implementing a new long-running query using *mdl-lonq* library constructs and utilities

The first step of the process is to consider existing state machines and if one of them can be used for the data source and query type. If not, a new state machine with an appropriate query chain must be implemented. As described in section 6.3.1, the state machine should not define the query but offer input parameters to configure it. Therefore, the *ProcessedLonqStateMachine* or *BaseLonqStateMachine* class can be subclassed. If the query results do not need further processing, the state machine construct can be instantiated and used as is. Otherwise, a compatible *LonqExtractionIterator* subclass must be implemented if one does not exist yet. The same applies to the *QueryProcessor* subclass. Both subclasses can be used to create a new Lambda handler or to extend an existing one using the *create_handler* function. They must be passed within a dictionary wrapped in a *Callable* to instantiate the desired processor at runtime. The Lambda function with this handler can then be handed to the defined state machine constructor and executed after the query chain.

With the state machine design of the LonqAPI, the input to a state machine execution defines the query and its parameters, as well as its processing by including the specific processor to execute and its parameters. The processor Lambda function can then instantiate the *LonqExtractionIterator* and *QueryProcessor* based on the input and process the query results.

7.3.2 Athena POM Query Implementation

To enable Athena queries, the *AthenaLonqStateMachine* class is implemented as a subclass of *ProcessedLonqStateMachine*. It implements the abstract query chain method and returns an *AthenaStartQueryExecution* task wrapped in a *Chain* object. A workgroup passed to the Athena query task defines the default S3 output location. The defined task does not specify the query but expects the *QueryString* and *QueryParameter* values in the state machine execution input. To set necessary permissions, the *AthenaLonqStateMachine* class overrides the *_set_state_machine_permissions* method and adds IAM statements to the state machine execution role. Since this class is not specific to the use case of the POM query, it is located in the *infrastructure* package of the *mdl-lonq* library.

Results of a *UNLOAD* query are stored in a S3 bucket in parquet format. To iterate over the generated files, the *S3ParquetIterator* is implemented as a subclass of the *LonqExtractionIterator*. It can be used with a list of keys or an S3 prefix to return the query results in chunks. The *read_parquet* function in the *s3* module from the *AWS SDK for pandas (awswrangler)* [117] is used to read the parquet files from S3 and return the results as a *pandas.DataFrame* object. The library function allows partial reading of a parquet file divided into parts, which can be used by passing its *chunked* parameter to the *S3ParquetIterator* constructor. Like the *AthenaLonqStateMachine* class, the *S3ParquetIterator* class is not specific to a use case and is located in the *data_processing* package of the *mdl-lonq* library.

As the next step, the query-specific *QueryProcessor* subclass must be implemented. For the Athena POM query, the *PomDataImperialProcessor* subclass enables the conversion of metric POM values to imperial units. It expects a *S3ParquetIterator* object as a constructor parameter and overrides the *_transform_element* method to integrate the existing *to_imperial* utility function from the *mdl_data_access* module. Results are again persisted as parquet files in S3.

To integrate the query result processing into the state machine execution, a Lambda function must be passed to *AthenaLonqStateMachine*. Therefore, the instantiation of the *S3ParquetIterator* and *PomDataImperialProcessor* objects is wrapped in a *Callable* that is passed to the *create_handler* function as a dictionary value. The dictionary key for the processor is set to *PomDataImperialProcessor* and can be used as the *Processor* key in the state machine execution input to trigger the processing after the query chain. This handler creation is documented in listing A.9 in the appendix. The returned handler function is then passed within a CDK *IFunction* object to the *AthenaLonqStateMachine* constructor to create the state machine. The *PomDataImperialProcessor* and created lambda handler are specific to the Athena POM query and, therefore, are located in the *lambda_handler* package of the *mdl-daas-history* subproject.

The resulting state machine can be used to execute Athena queries in general and is not specific to a S3 bucket or query type. However, the POM query state machine instance is only permitted to read from the POM data source S3 bucket and write to a defined result S3 bucket and prefix. Figure 7.4 shows the state machine instance visualized in the AWS Step Functions editor. The *PomDataImperialProcessor* within the processor Lambda function is query-specific and can only be used to convert metric POM values to imperial units. However, the dictionary passed to the Lambda handler creator function can be extended to add other processors. The necessary processor can be selected by passing the appropriate dictionary key in the state machine execution input.

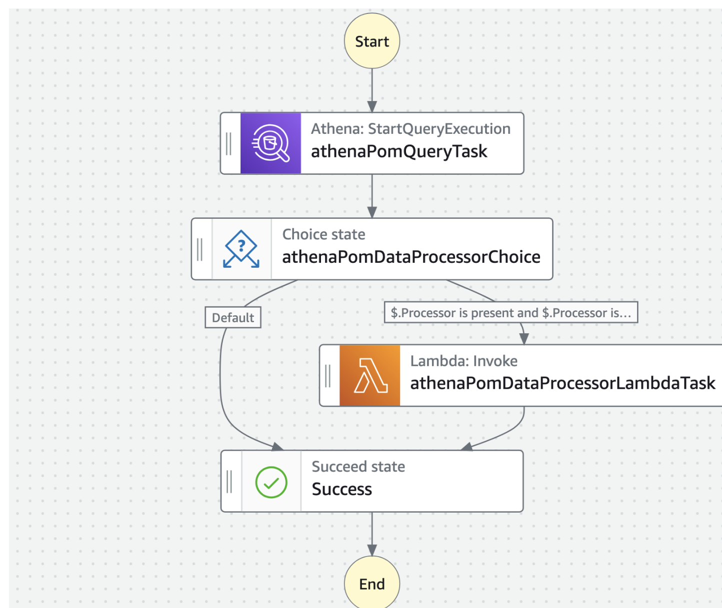

Figure 7.4: The Athena POM query state machine shown in the AWS Step Functions editor

The previous paragraphs simplify the implementation in some parts to focus on the essential aspects. For example, a *DataframeQueryProcessor* class is implemented that can be used to process query results as *pandas.DataFrame* objects. This class is the parent of the *PomDataImperialProcessor* class and implements the *_persist_element* method to store the results in S3 because this is not specific to POM data and can also be used for other queries.

7.4 mdl-daas-history Project Implementation

This section describes the implementation of the *mdl-daas-history* subproject introduced in section 7.1.2. To meet the requirements of the LonqAPI prototype, the project implements the infrastructure and logic using the *mdl-lonq* library and the Litestar REST API framework. The implementation process in section 7.3 implements the Athena POM query state machine construct and, therefore, is part of the *mdl-daas-history* subproject. The following sections explain the infrastructure definition of the application within one CDK stack and the implementation of the REST API Lambda function.

7.4.1 DaaS History API Infrastructure Stack

All application infrastructure is defined within one CDK stack called *DaasHistoryStack*. This follows the architecture shown in section 5.4 and design decisions in chapter 6. Configured settings fit the requirements of the LonqAPI prototype but can be adjusted to meet future requirements. The following paragraphs describe the essential parts of the stack implementation.

Result S3 Bucket: The CDK stack defines a new S3 bucket to store the query results. It is encrypted with an AWS Key Management Service (KMS) key created using existing utilities in the Data Layer implementation. The bucket is configured to allow only Secure Sockets Layer (SSL) encrypted connections and to block public access. Object versioning is disabled because changes to query results are not expected and would only increase storage costs. A lifecycle rule is defined to expire objects after 30 days to avoid unnecessary storage but keep them available for enough time to retrieve them.

Request Status DynamoDB Table: The CDK stack instantiates a new DynamoDB table to store the query request status information. The table has a partition key named *client_id* and a sort key named *query_id* to identify a query. The client_id is added because every query is associated with exactly one client and must be specified in LonqAPI requests, as described in section 6.1.3. To expire and delete old query status entries, the *expiration_time* field is set as the time to live attribute. It contains the expiration timestamp as seconds since the Unix epoch [72, Sec. 4.16]. Like the result S3 bucket, the DynamoDB table is encrypted using a new KMS key.

REST API and query result processor Lambda functions: Two Lambda functions are defined. Both build their own Docker image using the *DockerImageFunction* class and a Dockerfile based on the AWS Lambda Python 3.10 base image [118]. The first one is the REST API handler called by API Gateway. Its implementation is described in section 7.4.2. The function is configured with a 30-second timeout because the synchronous requests should be processed quickly. To do so, 3000 MB of memory are allocated to the function, which is more than necessary but should speed up the processing because CPU power is allocated proportionally to the memory size [79]. Necessary information like state machine Amazon Resource Names (ARNs) and the DynamoDB table name are passed to the function as environment variables. The second Lambda function is the query result processor called by the state machine if necessary. Its implementation is described as an example of the implementation process of a long-running query in section 7.3.2. For the LonqAPI prototype, this function iterates over the query results and converts the metric POM values to imperial units. Therefore, it is configured with the maximal timeout of 15 minutes and 4000 MB of memory, which experimentally is enough for Athena POM query results in chunks generated by the UNLOAD command. This processor function is defined to be extendable to implement other processing use cases in the future. In the context of this thesis, the function is only used for converting POM values to imperial units.

API Gateway: The REST API Lambda function is integrated into an API Gateway REST API. Compared to an HTTP API, REST APIs offer more features like API keys and caching [119]. The CDK *LambdaRestApi* class defines the API and integrate the Lambda

function. The API is deployed to a stage representing the environment, which is *dev* for the prototype.

Athena POM query state machine: The Athena POM query state machine construct is implemented as described in section 7.3.2. The result S3 bucket and the query result processor Lambda function are passed to its constructor as CDK *IBucket* and *IFunction* objects. Also passed are references to the POM data source for the Athena query, like the database and table name and the bucket to read from. The state machine construct is then built and added to the CDK stack, which is documented in listing A.7 in the appendix. The resulting state machine is shown in figure 7.4, visualized in the AWS Step Functions editor. To handle the execution of the state machine, the REST API Lambda function is granted permission to start and read the state machine execution.

7.4.2 REST API Lambda Function

The REST API Lambda function uses a Litestar application object and the Mangum ASGI adapter to implement the REST API endpoints. The implementation is separated into three main layers: the controller, service, and repository. This follows the three-layer architecture described by Fowler [120, pp. 19-22]. The controller layer defines the REST API endpoints and their request and response models. The service layer implements all business logic and uses the repository layer to access data or external services. Implemented classes and request handlers and their hierarchy within the layers are shown in figure 7.5 and described in the following paragraphs.

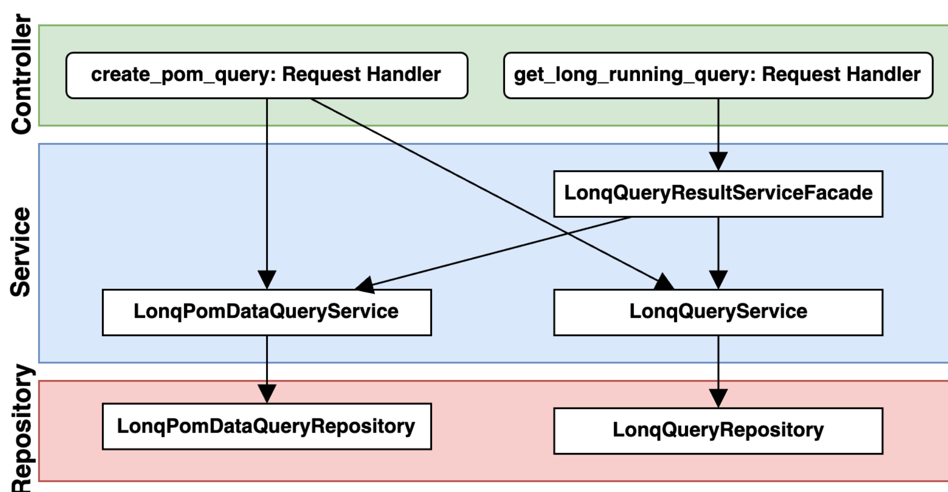

Figure 7.5: The implemented REST API calsses and request handlers, and their hierarchy in the layer architecture

The architecture is wrapped in a Litestar application object that is instantiated in the *app* module in the root package of the *mdl-daas-history* subproject. Litestar's dependency injection functionality is used to inject the layer objects into each other. Therefore, provider functions are implemented to instantiate the objects and register the dependencies in the application object. The REST API endpoints are defined in the *controller* package within the *query* and *pom_data_query* modules. Each controller module defines a *litestar.Router* with its endpoints, which are also registered in the Litestar application.

The *pom_data_query* module in the controller layer contains the endpoint to create a new Athena POM query, which is defined in section 6.1.3. This endpoint is implemented as the *create_pom_query* function and documented in listing A.10 in the appendix. The path and query parameters, as well as the service layer dependencies, are injected as parameters to the function. The *LonqPomDataQueryService* allows to start an Athena POM query with all necessary parameters. It returns a state machine execution ARN, and together with a generated query ID, the query with the status *RUNNING* is stored in the DynamoDB table using the *LonqQueryService*. For the *expiration_time* attribute, the current timestamp with an offset of one day is used. The new *LonqQuery* entity is returned as defined in section 6.1.3 in the response body with a *Location* header and the *201 Created* status code. The *UNLOAD* Athena query is defined in the *LonqPomDataQueryRepository* and passed as *QueryString* to the state machine execution input. If the *imperial* query parameter is set to *true*, the *Processor* key triggers the processing of the query results in the state machine. Since the state machine's output differs depending on whether the query is processed or not, the output type is also saved in the DynamoDB table. This is implemented in the *LonqQueryRepository* class and injected into the *LonqQueryService* class.

The query status polling endpoint, described in section 6.1.3, is implemented as the *get_long_running_query* function in the *query* module. To retrieve the current query status, the *LonqQueryResultServiceFacade* implements the Facade design pattern [115, pp. 185-193], [116, pp. 123-132] to hide the complexity of updating the query status and generating presigned S3 URLs. It uses the *LonqQueryService* to retrieve the persisted query entity and the *LonqPomDataQueryService* to retrieve results from S3 if the output was not processed. The status of the query is updated by getting the current state machine execution status. If this status turns to *SUCCEEDED*, a presigned S3 URL is generated for each result file and added to the query entity. The updated entity is then returned in the response body.

Litestar's internal exceptions are used to indicate errors in the controller layer and return the appropriate HTTP status codes, like the *ValidationException* in the *create_pom_query* handler. Litestar also validates and parses the client request variables based on the defined types and returns a 400 (Bad Request) status code if the request is invalid. To enhance the OpenAPI specification generated by the framework, the endpoints are documented using optional description parameters and types. Like the request parameters the response models are also specified by Python types and Pydantic models, and injected into the OpenAPI documentation.

7.5 Lambda Session Management

The AWS Lambda execution environment is reused for subsequent invocations. This means that after an initial invocation, the environment with its global variables is kept available for a certain time for further invocations. To reduce runtime and initialization time, the best practice is to initialize SDK clients and database connections outside the function handler [79]. Nevertheless, depending on the Lambda event, the function handler might only need specific resources, and not all possible connections should be

initialized. A solution is to initialize a session once needed and keep it for other invocations in the same execution environment. This is implemented in the *SessionManager* class as a shared module in the Data Layer monorepo and shown in listing A.11 in the appendix. The class is inspired by the Singleton pattern [115, pp. 127-134], [116, pp. 23-35] to provide simple access and reusability of an instance. However, it wraps not only one but multiple types of singleton sessions. The sessions are stored in a dictionary with the session type as the key and its object as the value. The single session objects are accessible through property methods. The *SessionManager* can be initialized globally in the application and used to access the session objects on demand. Throughout the LonqAPI implementation, it is used to initialize and access boto3 Software Development Kit (SDK) [121] clients and sessions for AWS services.

7.6 Summary

This chapter illustrated the implementation of the LonqAPI prototype and its two subprojects in the Data Layer monorepo, *mdl-lonq* and *mdl-daas-history*. The *mdl-lonq* subproject provides a library to facilitate the implementation of long-running queries in AWS Step Functions state machines. It defines extendable state machine constructs and a configurable query result processor Lambda handler. These library components can be used within a defined development process to implement new long-running queries. The implementation of the Athena POM query state machine and its result processor Lambda function is an example of this process. The *mdl-daas-history* subproject implements the LonqAPI prototype as a REST API using the created state machine construct and Lambda handler. It defines the AWS infrastructure and implements the two REST API endpoints for POM queries using the Litestar framework in a three-layer architecture. To facilitate session management in AWS Lambda functions, a *SessionManager* class is implemented as a utility and used throughout the LonqAPI implementation.

Chapter 8

Evaluation

This chapter aims to evaluate the LonqAPI prototype in terms of usability, efficiency, and extensibility. The evaluation is based on the requirements and user stories in section 4.3 and the research questions and objectives in Chapter 1. It is divided into three sections presenting parts of the evaluation, and the last section concludes. First, the usability and performance of the REST API is evaluated. The second section focuses on the efficiency of the long-running query processing in AWS Step Functions state machines. This is followed by an evaluation of the extensibility of the prototype with the implementation of a new query data source. The last section summarizes the evaluation results and reviews the research questions and objectives.

The evaluation is carried out through manual tests and measurements. However, automated tests are also integrated into the development process of the whole Data Layer monorepo and cover the LonqAPI implementation as well. These consist of the execution of unit tests and static code analysis. Unit tests are implemented for each Python module and are executed by the pytest framework [122]. Static code analysis is performed by Pylint [123] and Mypy [124] for type checking.

8.1 API Evaluation

This section evaluates the LonqAPI REST API from the user's perspective. Its usability is evaluated by a test client implementation, and a load test evaluates its performance.

8.1.1 Test Client Implementation

A test client is developed to evaluate the REST API usability. It is implemented as a Python script using the *requests* library [125] to send HTTP requests to the API. Depending on how code is formatted and how lines are counted, the number of lines of code is between 20 and 50, including imports and configuration parameters. The structure can also be seen in figure 8.1, which shows the flowchart diagram of its implementation.

In general, a client has to implement three different requests: one to start a query, one

to get the query status, and one to get the query results by the provided presigned S3 URLs. However, only the first request has to be configured with path and query parameters. The other two are retrieved from the responses. Two loops are used. The first one implements the client polling for the query status until it is completed or failed. The second loop iterates over the result S3 URLs and downloads the result files. Clients need to be aware of failures, which is why the script can terminate if an unsuccessful HTTP or query status code is returned.

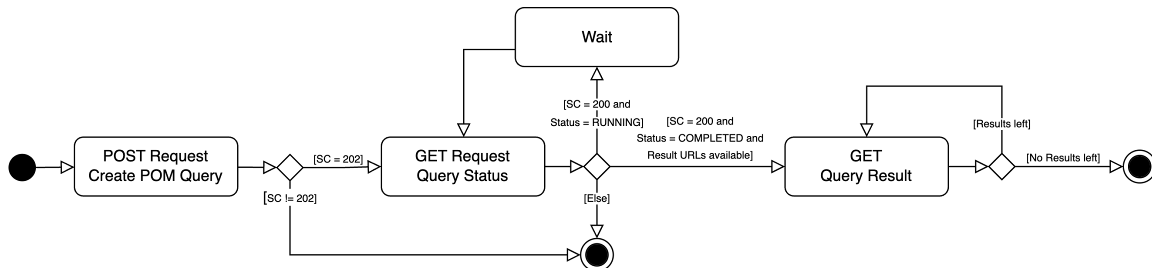

Figure 8.1: The test client implementation flowchart diagram

The relatively simple implementation structure is not only specific to Python but can be implemented in other popular programming languages, especially because libraries for HTTP requests are available for most of them. Some tools and libraries are presented in webpages like [48]. The passing of URLs in the responses reduces complexity and request configuration. This is also shown by the low number of lines of code. The client needs to handle different status codes and errors, which leads to more branching. However, detailed error messages can reduce debugging effort on failures.

8.1.2 API Performance Evaluation

One requirement of BHS to the LonqAPI is to support a user load of up to 10000 requests per minute, not including the initiation of long-running queries. Therefore, AWS API Gateway and the request-handling Lambda function must handle this load. This is evaluated by a load test using Apache JMeter [126].

The test is configured to send 11000 requests per minute as target throughput within 16 threads in one minute to the API query resource endpoint. In the path parameter, the ID of an existing query is used to retrieve its status. Therefore, the test does not start new queries but only retrieves the query status. The test also applies load to the DynamoDB table used to store the query metadata. First, a request goes to API Gateway and is then handled by the Lambda function implemented in Python using the Litestar framework [111]. The Lambda function retrieves the query metadata from the DynamoDB table and returns the query status as a JSON response. To simulate the API in production, the API is called manually before the test to ensure that the initial Lambda function cold start is not included.

As a result, 10179 requests are sent. All of them are successful and return a 200 HTTP status code. The response times are summarized in figure 8.2. Average and median response times are below 100 milliseconds. 99% of the requests are still answered within 256 milliseconds. The maximum response time is 1064 milliseconds. The AWS Lambda

metrics show that the function has 16 concurrent executions at its peak caused by the 16 threads of the load test.

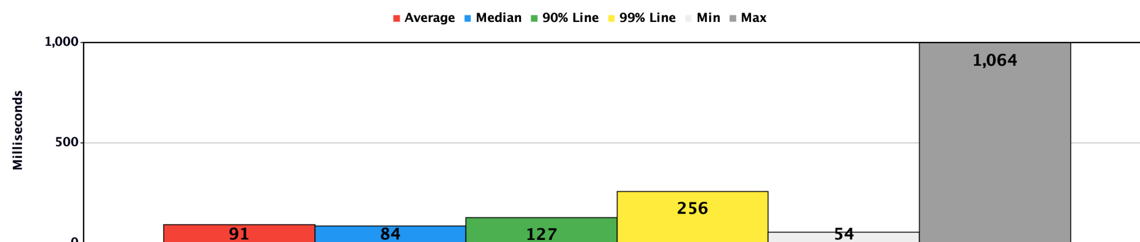

Figure 8.2: Response times of the API for 10179 requests to the query resource endpoint in milliseconds

These results show that the API can handle the required 10000 requests per minute. However, the test does not include the initiation of long-running queries, which can increase the execution time of the Lambda function. The load test, in general, shows good response time results. The maximum response time of more than one second is still acceptable for the use case, especially because 99% of the requests are answered within 256 milliseconds. It can be assumed that it is caused by a cold start of the Lambda function for concurrent execution. This matches the operator guide *Lambda execution environments* section in the AWS Lambda documentation [127], which states that cold starts take under 100 milliseconds to over one second.

8.2 Query Execution Infrastructure Efficiency Evaluation

This section aims to evaluate the efficiency of the AWS infrastructure used by the LonqAPI. It is divided into the inspection of the infrastructure execution time and costs overhead caused by the LonqAPI. Both are based on the Athena POM query described in section 4.2.3. In this context, overhead refers to the time and costs caused by the LonqAPI, excluding the actual query execution in Athena.

8.2.1 Query Execution Time Overhead

To measure the time overhead from the request until the query results are available, the client script from section 8.1.1 is used. Therefore, a minimal query is executed, which only takes about one second to execute in Athena. The timestamp is saved before the request is sent, and the remaining times are looked up in the corresponding AWS service consoles. The processing time of the Lambda function within the state machine is highly dependent on the query result size and number of result files. Also, the implementation of the utilized *to_imperial* function, described in section 4.2.3, is not part of this thesis. Therefore, the time of the query result processor Lambda function execution is not considered in the overhead and is only mentioned in the following example.

Timestamps relative to the client request on a local machine are collected for an example query execution and shown in table 8.1. Only relevant timestamps focused on the state machine and query execution are included. The results show that it takes about half a

second until the Athena query is submitted, which, in this case, completes relatively fast after another 1.3 seconds due to the minimal query. However, it takes nearly a minute until the state machine continues after the Athena query is completed. This delay is neither part of the Athena execution nor an exception but also observed in other tests. The other state transition times summed up are about half a second. The processor Lambda function finishes relatively fast in about 174 milliseconds because the query does not return any data, and no result files need to be processed. Altogether, the query execution overhead in this example is 59.532 seconds. For this, the Athena query and the processor Lambda function execution time are deducted from the total time of 60.985 seconds.

Timestamp Description	Relative Timestamp (Seconds)
POST request	0.000
State machine started	0.139
Athena query submitted	0.515
Athena query completed	1.794
State machine continued	60.646
Processor Lambda function started	60.742
Processor Lambda function completed	60.916
State machine completed	60.985

Table 8.1: The Athen query execution timestamps of the LonqAPI infrastructure relative to the POST request

These results can not be compared quantitatively since no other query execution infrastructure is implemented. However, using the Fargate query execution service approach described in section 5.3.4, the query execution time overhead is expected to be lower. This is because the Fargate task can execute the query synchronously, and the delay within the state machine execution after the query completion is absent. This is a disadvantage of the implemented approach based on unexpected behavior of the AWS Step Functions service in combination with the asynchronous Athena query execution and should be considered in future development.

To retrieve the query results, the client must poll the query status endpoint until the query is completed and then download the result files. This also causes a time overhead, which depends on the client's polling interval. In the best case, the additional overhead is only the response times of the query status endpoint and the result S3 URLs. In the worst case, the overhead is also increased by one polling period. This is a disadvantage of the polling pattern described in section 3.1.2 and must be considered when using the API in applications.

8.2.2 Query Cost Overhead

To evaluate the cost overhead of the query execution infrastructure, the AWS Pricing Calculator [128] is used. The costs are highly dependent on the infrastructure usage, and since no concrete numbers are available, only estimations can be made. Also, the variety of possible configurations makes it difficult to compare the infrastructure costs

with other approaches. Therefore, the following gives only an overview of possible costs of LonqAPI infrastructure components and their alternatives.

REST API Costs

The REST API is implemented using an AWS API Gateway REST API and AWS Lambda. Both are billed per request, and Lambda also per execution time and memory allocation. The API Gateway is billed with 3.70 $ per million requests. The Lambda function handling the requests is configured with 3 GB in the x86 architecture, and executions are expected to take between 100 and 300 milliseconds on average. This is higher than the measured average execution time in section 8.1.2 because the test only retrieves existing queries and does not start new ones. Less frequent requests could also increase the average execution time. With 300 milliseconds and 3 GB, the Lambda function is billed with 14.85 $ per million requests.

An EC2 instance could be used as an alternative to both services. The cheapest instance type with more than 3 GB of memory is the *t4g.medium* instance with 4 GB. As an *On-Demand* instance, it is billed with 28.03 $ per month. This is equivalent to the costs of more than 1.5 million requests per month with API Gateway and Lambda. Using a savings plan or a spot instance can reduce EC2 costs.

The comparison shows that the API Gateway and Lambda approach is a sufficient and cost-efficient solution for the prototype. However, costs must be considered in future production use cases, and other approaches, like EC2 instances, can be evaluated.

Query Status DynamoDB Table Costs

The query status DynamoDB table stores the query metadata and uses the on-demand standard table pricing model. A single entry, including result URLs, is expected to be less than 2 KB. With this size, one GB can store about 500000 entries for a cost of 0.306 $ per month. These costs are expected to be less since the entries are configured to expire after one day and are deleted automatically. One million reads and 100000 writes per month can be a realistic estimation for the number of requests mentioned for REST API in the previous section. Reads are expected to be more frequent because of the implemented polling pattern. This results in 0.30 $ per month. Therefore, the costs of the DynamoDB table are expected to be negligible in this context and prove the choice of the NoSQL database in terms of cost-efficiency.

Query Result S3 Bucket Costs

Results are stored in a S3 bucket and will be deleted automatically after 30 days. As described in section 5.3.2, S3 is designed to handle large data volumes. Other AWS services like RDS or DynamoDB handle other use cases. Therefore, and because no production experience is available, costs can hardly be evaluated. However, the storage of one TB per month is billed with 25.09 $. One million PUT, COPY, POST, or LIST requests to S3 cost 5.40 $. One million GET or SELECT requests cost 0.43 $. Inbound data transfer is free, and outbound data transfer of one TB is billed with 92.16 $. These costs

need to be considered when using the DaaS API in production. However, optimizations like a lower lifetime of the results or limits for the executed queries can reduce the costs.

Query Execution Costs

The query execution is implemented using an AWS Step Functions state machine. This service is billed by its number of executions and state transitions. The developed state machine is configured with five state transitions, including the optional processor Lambda state. 100000 executions with five state transitions are billed with 12.40 $ per month.

This example calculation can be compared with the costs of the Fargate query execution service described in section 5.3.4 as an alternative approach. Its costs depend on the task execution time, which includes Athena's query time. With an estimated average execution time of one minute, which is the minimal priced time, 1 vCPU, and 3 GB of memory, 100000 on-demand task executions are billed with 123.78 $.

This shows that the Step Functions state machine is more cost-efficient than a Fargate task by a factor of 10 in this example calculation. Its advantage is that it does not use computing resources while the query is executed, which reduces the cost overhead compared to other approaches.

Query Result Processing Costs

A Lambda function optionally processes the results of the Athena POM query to convert the data format from metric to imperial. It is integrated as a state in the Step Functions state machine but can be replaced by other approaches. Therefore, its costs are evaluated separately. The Lambda function execution time depends on the query result size and number of result files. 1000 executions with the configured 3 GB of memory in the x86 architecture and an average execution time of one minute are billed with 3.00 $.

Alternatively, a Fargate task can be used to process the query results. 1000 AWS Fargate on-demand one-minute task executions with 1 vCPU and 3 GB memory are billed with 1.24 $. The costs of both approaches change proportionally to the number of executions and their average runtime, but a Fargate task has a minimum billed execution time of one minute. Lambda is charged per millisecond without a minimum.

This shows that Lambda can be more cost-efficient for shorter executions than the minimal one-minute Fargate billing time. Fargate is more cost-efficient for longer executions. Since query results are expected to be large, Fargate can be a better choice and can be evaluated to replace the Lambda function in future development, depending on the concrete use case in production.

Costs Summary

Table 8.2 summarizes possible monthly costs per AWS service, volume, and action. The numbers are based on one million requests, 100000 query executions, and 1000 result processing executions. The query result size is assumed to be one TB per month in

storage. This shows that especially the S3 outbound data transfer costs are higher than the others and need to be considered. However, these are just example calculations to give an overview of the costs and the relevant services in this context.

AWS Service	Volume and Action	Costs per Month
API Gateway	1 million API requests	3.70 $
Lambda	1 million API request handler executions	14.85 $
DynamoDB	100000 query status entries of 2 KB	0.06 $
DynamoDB	1 million reads and 100000 writes	0.45 $
S3	1 TB query result storage	25.09 $
S3	1 million PUT, COPY, POST, or LIST requests	5.40 $
S3	1 million GET or SELECT requests	0.43 $
S3	1 TB outbound data transfer	92.16 $
Step Functions	100000 query state machine executions	12.40 $
Lambda	1000 result processing executions	3.00 $

Table 8.2: Exemplary monthly costs per AWS service, volume, and action for the LonqAPI infrastructure

8.3 Extensibility Evaluation Example

This section evaluates the extensibility of the LonqAPI prototype by implementing a new query data source. Therefore, DynamoDB was chosen because it is already integrated into the BHS Data Layer and the implemented query status table can be used as the concrete data source. The goal is to retrieve all long-running queries from the DynamoDB table for the requesting client. This is a short query regarding execution time and would not be considered long-running. However, for this evaluation, it is sufficient to test the extensibility of the prototype.

To enable the integration of DynamoDB into the LonqAPI, the implementation is extended by the classes and functions listed in table 8.3. Not included is the registration of additional dependencies in the Litestar application and the *DynamoDBLonqStateMachine* class instantiation in the *DaasHistoryStack* class.

The *DynamoDBLonqStateMachine* is implemented as a new state machine construct class to execute the DynamoDB *Query* action [129]. Because state outputs are limited to 256 KB, the results must be directly persisted outside the state machine. The synchronous DynamoDB query does not support this by default. Therefore, the *MapLonqStateMachine* is implemented and inherited by the *DynamoDBLonqStateMachine* to use the *Map* state and its *ResultWriter* as documented in the AWS Step Functions developer guide under *Input and Output Processing in Step Functions* [87]. With the query wrapped inside the *Map* state, the output is directly written to S3 to avoid the 256 KB limit. The query state is implemented as a *CallAwsService* state with appropriate parameters, including the *KeyConditionExpression*. This state allows the query execution without consuming computing resources. The query output is written to one manifest file that contains S3 URLs of the actual query result files. Both are in the JSON format. The *ResultWri-*

Class / Function Name	Purpose
MapLonqStateMachine	Generic usage of a Map state and its ResultWriter in a state machine
DynamoDBLonqStateMachine	Generic query state implementation for DynamoDB state machines
ResultWriterIterator	Generic LonqExtractionIterator for chunked extraction of Map state results
DynamoDBQueryProcessor	Generic QueryProcessor to normalize DynamoDB query results and write them as Parquet to S3
build_dynamodb_query_processor	Creator function for the DynamoDBQueryProcessor instantiation at Lambda function runtime
create_dynamodb_query	Litestar request handler function to create the DynamoDB query
LonqDynamoDBQueryService	Service layer class for business logic (none in this case)
LonqDynamoDBQueryRepository	Repository layer class to start the state machine with the DynamoDB query

Table 8.3: The classes and functions implemented to integrate DynamoDB into the LonqAPI

terIterator is implemented to read the manifest file and iterate over the result file data. The *DynamoDBQueryProcessor* is implemented to normalize the data because the result items include the DynamoDB data types like *S* for a string or *N* for a number. Therefore, an existing utility function is used from the Data Layer monorepo. This processor class is appended to the configuration of the existing Lambda function to be used by the name *DynamodbQueryProcessor* set in the state machine input. The concrete query is implemented in the repository layer *LonqDynamoDBQueryRepository* class and called by the *LonqDynamoDBQueryService*. This service layer class only passes the query parameters to the repository since no business logic is necessary in this case. The *create_dynamodb_query* function defines the request endpoint and is registered as a request handler in the Litestar application.

Six new classes and several functions are implemented to integrate DynamoDB as a query source into the LonqAPI. This is a relatively high number for a simple query. However, the implementation is generic and can also be used for other data sources. The REST API layer architecture allows easy integration, and Litestar provides a simple way to register new request handlers and dependencies. Since the query resource endpoint is used for all queries, only one new endpoint is implemented. The implemented template method pattern in the processor and state machine classes, described in section 7.2, facilitates the extensibility by reusing existing code and providing a simple structure. Also, the easy extension of the processor Lambda function by expanding its configuration is an advantage of the implemented architecture. However, although base classes are used for the new state machine construct, the integration causes trial and error to pass the correct inputs and outputs between the states. Especially the necessary *Map* state and its *ResultWriter* can be complicated.

This shows that new query sources can be integrated into the LonqAPI and the implementation supports the extension, but effort is required depending on the concrete scenario. An advantage is the generic library that can be extended during the development process for reusability. These evaluation results are summarized in table 8.4.

Pros	Cons
Well-organized REST API layer architecture	Complexity by *Map* state and its *ResultWriter*
Low boilerplate code due to Litestar framework	State input/output passing trial and error
Endpoint reuse for all queries	
Reusability of template method pattern	
Easy extension of processor Lambda function	
Reusability of generic library implementations	

Table 8.4: Evaluation of the DynamoDB integration into the LonqAPI

8.4 Conclusion

The research objective of this thesis is to develop a consistent DaaS interface for long-running queries on a Data Layer in the AWS Cloud. The challenge is to enable asynchronous client-server communication using a suitable interface technology and to process long-running queries and large data results decoupled from the interface. To face this challenge, the LonqAPI as a DaaS interface in the Data Layer at BHS is developed as a prototype. It aims to provide a reusable and extensible framework concept for similar scenarios in the AWS Cloud to achieve effective design and implementation in terms of interface technology and infrastructure architecture.

The following paragraphs summarize the evaluation results and review the research questions to answer if the research objective is achieved.

Research Question 1 (RQ1): Which technologies and architectures are available to design and implement a DaaS interface providing data from long-running queries in the AWS Cloud? Fundamentals to answer this research question are presented in the state of the art chapter (3). It describes asynchronous communication and correlation techniques, including client-server patterns based on conversation metadata. Also, popular API technologies are characterized. Based on the requirements of the LonqAPI, the REST API technology with the polling pattern is selected and embedded in an architecture in the cloud-native Data Layer at BHS. The infrastructure uses AWS API Gateway and Lambda to implement the REST API. As alternatives, the services EC2 and Fargate are introduced. Results are provided using the S3 presigned URL feature. The REST API is designed with considerations about the Lambda request handling. This resulted in a monolithic Lambda function with the Python Litestar framework. The implementation defines infrastructure using AWS CDK and the request-handing Lambda function in a three-layered architecture.

Regarding the implemented prototype, the REST API usability and performance evaluations show good results with a simple client implementation and a low response time to requests. Cold starts of the Lambda function are observed, which can increase the response time of infrequent or concurrent requests. Cost overhead demonstrates the API Gateway and Lambda approach as a sufficient and cost-efficient solution for the prototype. The drawback of the polling pattern is the additional time overhead because the client does not receive the results automatically.

Regarding the presented overview of technologies and architectures, this thesis only summarizes the fundamentals and focuses on the REST API technology beginning with

chapter 5. Only the polling pattern is designed, implemented, and evaluated. Therefore, the answer to this research question is limited and can be extended by further research.

Research Question 2 (RQ2): How can long-running queries be efficiently processed to provide data based on Athena queries for a DaaS interface using AWS Cloud services? The LonqAPI prototype is designed and implemented to answer this research question. Based on insights from research about ETL processes, the Athena POM query is integrated within an *UNLOAD* statement into an AWS Step Functions state machine decoupled from the DaaS interface. The transformation and loading steps are implemented as a Lambda function, and results are stored in S3.

As described in section 5.3.4, Athena does not need to be integrated into a state machine for this question, but this allows to give answers to research question RQ3. The evaluation shows that the state machine is a sufficient and cost-efficient solution. However, it causes an execution time overhead of about one minute in this implementation. Depending on the use case, Lambda is no suitable choice for large query results, and services like Fargate can be evaluated as an alternative. S3 is designed to handle big data volumes, but optimization may be necessary in large-scale production use cases.

Altogether, the LonqAPI answers this research question for the case example at BHS, but other requirements may lead to different solutions.

Research Question 3 (RQ3): How can the system be designed to be flexible and extensible to further or changing long-running queries and data sources? Architecture, design, and implementation of the LonqAPI are focused on answering this question. Within the AWS architecture, a generic state machine definition allows the integration of new query sources and processing steps. The REST API is designed to be flexible and extensible using a monolithic Lambda, a three-layered Litestar application architecture, and a generic query resource. The separation in library and application implementation aims to increase reusability.

Next to Athena, DynamoDB is integrated as a new query source to evaluate the extensibility. Results show that the architecture and implementation support the extension without changing the system's structure, but depending on the concrete scenario, effort is required to define the state machine. However, the generic library allows the reuse of development results.

This answers the research question for the system of the LonqAPI prototype. However, it is still in a proof-of-concept state, and future work is necessary to evaluate its extensibility for other scenarios. The answer is also limited to the case example use case. It does not include other system designs or requirements like joining query results.

Chapter 9

Summary and Future Work

The following sections summarize the results of this thesis and provide an outlook on future work.

9.1 Summary

The introduction (1) presented the necessity of data availability to benefit from data-driven innovations and business decisions in the IIoT big data environment. To fill this lack at BHS Corrugated, as the case example in this thesis, the LonqAPI was developed as a DaaS interface prototype for long-running queries in its Machine Data Layer.

The research included asynchronous communication and API technologies to make data accessible to clients. Based on BHS requirements, the architecture of the LonqAPI as a REST API in the AWS cloud was constructed from necessary components and their interactions. The implementation was further designed and described with a focus on extensibility and usability.

The evaluation (8) showed that the LonqAPI is a suitable approach for the use case at BHS. A generic library implementation enables access to data based on the Athena POM query and facilitates the integration of new queries and data sources. With API Gateway, Lambda, Step Functions state machines, S3, and DynamoDB, the LonqAPI is a cost-efficient and flexible cloud-native prototype solution. The REST API polling pattern proved its usability for client applications, and the architecture is scalable to handle the required load.

The answers to this thesis's research questions and objectives were evaluated and summarized. However, the LonqAPI is still in a proof-of-concept state and requires further evaluation and development to be used in production.

9.2 Future Work

Based on the results of this thesis, the following future work is suggested:

As mentioned in the evaluation (8), costs and performance are hard to predict and should be monitored to gain better insights. Therefore, the LonqAPI should be evaluated in a production environment. New use cases and requirements can be identified and implemented to find potential improvements and increase its generic usability.

Cloud computing poses challenges for security, as presented by Pakmehr et al. [8], and must be considered for the LonqAPI in production. This thesis does not focus on security aspects, and further research must be conducted to ensure a secure and reliable DaaS interface.

The implemented polling pattern for asynchronous communication has the drawback that the client does not receive results immediately but has to poll for it. This can be improved by extending the API and its underlying infrastructure to support the callback pattern described in section 3.1.2.

Beginning with architectural decisions in chapter 5, the LonqAPI was designed as a REST API. However, GraphQL is a promising alternative that can be evaluated and compared to it. Especially its subscription feature could benefit the LonqAPI with asynchronous communication at transport level. Patterns based on conversation metadata can still be applied to GraphQL to increase client usability.

The query execution and its processing are crucial parts of the LonqAPI but are limited to the relatively simple requirements of the case example for this thesis. However, the developed concept provides the potential to integrate more advanced functionality within the query execution and processing AWS Step Functions state machine. This can include the combination of multiple queries, data sources, and other services, like machine learning, to gain more value from data. Focused on the long-running query execution, the LonqAPI does not support result caching. This can be implemented to reduce costs and improve performance.

Bibliography

[1] Statista. "IoT connected devices by vertical 2030," Statista, [Online]. Available: https://www.statista.com/statistics/1194682/iot-connected-devices-vertically/ (visited on 11/14/2023).

[2] S. Balaji, K. Nathani, and R. Santhakumar, "IoT Technology, Applications and Challenges: A Contemporary Survey," *Wireless Personal Communications*, vol. 108, no. 1, pp. 363–388, 2019. DOI: 10.1007/s11277-019-06407-w.

[3] N. Sharma, "Evolution of IoT to IIoT: Applications & Challenges," *International Conference on Innovative Computing & Communications (ICICC)*, 2020. DOI: 10.2139/ssrn.3603739.

[4] M. Bansal, I. Chana, and S. Clarke, "A Survey on IoT Big Data: Current Status, 13 V's Challenges, and Future Directions," *ACM Computing Surveys*, vol. 53, no. 6, 131:1–131:59, Dec. 6, 2020. DOI: 10.1145/3419634.

[5] C. Yang, Q. Huang, Z. Li, K. Liu, and F. Hu, "Big Data and cloud computing: Innovation opportunities and challenges," *International Journal of Digital Earth*, vol. 10, no. 1, pp. 13–53, 2017. DOI: 10.1080/17538947.2016.1239771.

[6] The Packaging Portal. "About BHS Corrugated," [Online]. Available: https://www.thepackagingportal.com/suppliers/bhs-corrugated/ (visited on 11/14/2023).

[7] E. Begoli and J. Horey, "Design Principles for Effective Knowledge Discovery from Big Data," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 215–218. DOI: 10.1109/WICSA-ECSA.212.32.

[8] A. Pakmehr, A. Aßmuth, C. P. Neumann, and G. Pirkl, "Security Challenges for Cloud or Fog Computing-Based AI Applications," *CoRR*, vol. abs/2310.19459, 2023. DOI: 10.48550/ARXIV.2310.19459. arXiv: 2310.19459.

[9] K. Taylor-Sakyi. "Big Data: Understanding Big Data." arXiv: 1601.04602. (2016), preprint.

[10] K. K. Patel, S. M. Patel, and P. G. Scholar, "Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, Application & Future Challenges," *International Journal of Engineering Science and Computing*, vol. 6, no. 5, 2016.

[11] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial internet of things (IIoT): An analysis framework," *Computers in Industry*, vol. 101, pp. 1–12, 2018. DOI: 10.1016/j.compind.2018.04.015.

[12] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Special Publication 800-145, 2011. DOI: 10.6028/NIST.SP.800-145.
[13] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. DOI: 10.1109/MCC.2017.4250939.
[14] AWS. "Overview of Amazon Web Services," [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html (visited on 11/11/2023).
[15] Statista, "Infrastructure as a Service (IaaS)," Statista. [Online]. Available: https://www.statista.com/study/31316/infrastructure-as-a-service-statista-dossier/ (visited on 11/11/2023).
[16] S. Rajesh, S. Swapna, and P. Shylender Reddy, "Data as a Service (Daas) in Cloud Computing," *Global Journal of Computer Science and Technology*, vol. 12, no. B11, pp. 25–29, 2012. [Online]. Available: https://computerresearch.org/index.php/computer/article/view/286.
[17] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications* (Data-Centric Systems and Applications). Springer Berlin Heidelberg, 2003, ISBN: 978-3-540-44008-6. DOI: 10.1007/978-3-662-10876-5.
[18] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 263–273. DOI: 10.1145/2872427.2883029.
[19] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How Does Web Service API Evolution Affect Clients?" In *2013 IEEE 20th International Conference on Web Services*, IEEE, 2013, pp. 300–307. DOI: 10.1109/ICWS.2013.48.
[20] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*. O'Reilly Media, 2012, ISBN: 978-1-4493-0892-6.
[21] S. Krompass, H. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper, "Managing long-running queries," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ACM, 2009, pp. 132–143. DOI: 10.1145/1516360.1516377.
[22] P. A. Laplante and S. J. Ovaska, *Real-Time Systems Design and Analysis*, 4th ed. Wiley-IEEE Press, 2011, ISBN: 978-0-470-76864-8.
[23] A. Reuther, C. Byun, W. Arcand, *et al.*, "Scalable system scheduling for HPC and big data," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 76–92, 2018. DOI: 10.1016/j.jpdc.2017.06.009.
[24] A. Podelko, "Multiple Dimensions of Performance Requirements," in *33rd International Computer Measurement Group Conference*, Computer Measurement Group, 2007, pp. 325–334.
[25] U. Zdun, M. Voelter, and M. Kircher, "Design and Implementation of an Asynchronous Invocation Framework for Web Services," in *Web Services - ICWS-Europe 2003*, Springer Berlin Heidelberg, 2003, pp. 64–78. DOI: 10.1007/978-3-540-39872-1_6.
[26] M. Brambilla, S. Ceri, M. Passamani, and A. Riccio, "Managing Asynchronous Web Services Interactions," in *Proceedings IEEE International Conference on Web Services*, IEEE, 2004, pp. 80–87. DOI: 10.1109/ICWS.2004.1314726.

[27] O. Ehab. "Synchronous VS Asynchronous Operations," [Online]. Available: https://www.linkedin.com/pulse/synchronous-vs-asynchronous-operations-omar-ehab (visited on 11/23/2023).

[28] A. Banks, J. Challenger, P. Clarke, *et al.*, "HTTPR Specification," version 1.1, *IBM Software Group*, 2002. [Online]. Available: https://xml.coverpages.org/IBM-ws-httprspec.pdf.

[29] N. Ivaki, N. Laranjeiro, and F. Araujo, "A survey on reliable distributed communication," *Journal of Systems and Software*, vol. 137, pp. 713–732, 2018. DOI: 10.1016/j.jss.2017.03.028.

[30] A. Lombardi, *WebSocket: Lightweight Client-Server Communications*. O'Reilly Media, 2015, ISBN: 978-1-4493-6925-5.

[31] HiveMQ Team. "MQTT Essentials: Part 3." (2019), [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/ (visited on 11/16/2023).

[32] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Middleware 2000*, J. Sventek and G. Coulson, Eds., Springer Berlin Heidelberg, 2000, pp. 208–230. DOI: 10.1007/3-540-45559-0_11.

[33] M. Voelter, M. Kircher, U. Zdun, and M. Englbrecht, "Patterns for Asynchronous Invocations in Distributed Object Frameworks," in *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPlop 2003)*, 2003, pp. 269–284. [Online]. Available: http://eprints.cs.univie.ac.at/2392/.

[34] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM*, vol. 35, no. 2, pp. 114–131, 2003. DOI: 10.1145/857076.857078.

[35] R. Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," in *Proceedings First International Conference on Peer-to-Peer Computing*, IEEE, 2001, pp. 101–102. DOI: 10.1109/P2P.2001.990434.

[36] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000, vol. 2, ISBN: 978-0-471-60695-6.

[37] M. Richards, *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*. O'Reilly Media, 2017, vol. 3, ISBN: 978-1-4919-2424-2.

[38] P. Sbarski, Y. Cui, and A. Nair, *Serverless Architectures on AWS*. Simon and Schuster, 2022, vol. 2, ISBN: 978-1-61729-542-3.

[39] S. H. A. El-Sappagh, A. M. A. Hendawi, and A. H. El Bastawissy, "A proposed model for data warehouse ETL processes," *Journal of King Saud University - Computer and Information Sciences*, vol. 23, no. 2, pp. 91–104, 2011. DOI: 10.1016/j.jksuci.2011.05.005.

[40] S. Q. A. Al-Rahman, E. H. Hasan, and A. M. Sagheer, "Design and implementation of the web (extract, transform, load) process in data warehouse application," *IAES International Journal of Artificial Intelligence*, vol. 12, no. 2, pp. 765–775, 2023. DOI: 10.11591/ijai.v12.i2.pp765-775.

[41] K. Krishnan, *Building Big Data Applications*. Academic Press, 2020, ISBN: 978-0-12-815746-6. DOI: 10.1016/B978-0-12-815746-6.00002-8.

[42] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Publication, University of California, Irvine, 2000. [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[43] B. De, *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, 2023, ISBN: 979-8-8688-0054-2. DOI: 10.1007/979-8-8688-0054-2_9.

[44] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," in *Proceedings of the 22nd International Conference on Software Engineering*, ser. ICSE '00, ACM, 2000, pp. 407–416. DOI: 10.1145/337180.337228.

[45] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "Can GraphQL Replace REST? A Study of Their Efficiency and Viability," in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, 2021, pp. 10–17. DOI: 10.1109/SER-IP52554.2021.00009.

[46] P. Kaur. "10 Most Popular Frameworks For Building RESTful APIs." (2023), [Online]. Available: https://www.moesif.com/blog/api-product-management/api-analytics/10-Most-Popular-Frameworks-For-Building-RESTful-APIs/ (visited on 11/27/2023).

[47] I. Wootten. "The Best Python HTTP Clients." (2022), [Online]. Available: https://www.scrapingbee.com/blog/best-python-http-clients/ (visited on 11/27/2023).

[48] P. Parthiban. "REST Client Made Easy: Exploring Top Libraries Across Languages." (2023), [Online]. Available: https://www.atatus.com/blog/rest-client-libraries-and-tools/ (visited on 11/27/2023).

[49] AWS. "Amazon API Gateway," [Online]. Available: https://aws.amazon.com/api-gateway/ (visited on 11/27/2023).

[50] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6455.txt.

[51] A. Deveria. "Can I use... | Web Sockets," [Online]. Available: https://caniuse.com/websockets (visited on 11/27/2023).

[52] Scaledrone. "WebSocket Libraries," [Online]. Available: https://www.scaledrone.com/websockets/ (visited on 11/27/2023).

[53] The GraphQL Foundation. "GraphQL Foundation," [Online]. Available: https://graphql.org/foundation/ (visited on 11/28/2023).

[54] The GraphQL Foundation. "Learn GraphQL," [Online]. Available: https://graphql.org/learn/ (visited on 11/28/2023).

[55] Apollo Graph. "Subscriptions - Get real-time updates from your GraphQL server," [Online]. Available: https://www.apollographql.com/docs/react/data/subscriptions/ (visited on 11/28/2023).

[56] The GraphQL Foundation. "GraphQL Code Libraries, Tools and Services," [Online]. Available: https://graphql.org/code/ (visited on 11/28/2023).

[57] AWS. "AWS AppSync," [Online]. Available: https://aws.amazon.com/appsync/ (visited on 11/28/2023).

[58] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984. DOI: 10.1145/2080.357392.
[59] The Linux Foundation. "About gRPC," gRPC, [Online]. Available: https://grpc.io/about/ (visited on 11/30/2023).
[60] The Linux Foundation. "Documentation - gRPC," gRPC, [Online]. Available: https://grpc.io/docs/ (visited on 11/30/2023).
[61] I. G. Buzhin, A. Y. Derevyankin, V. M. Antonova, A. P. Perevalov, and Y. B. Mironov, "Comparative Analysis of REST and gRPC Used in the Monitoring System of Communication Network Virtualized Infrastructure," *T-Comm*, vol. 17, no. 4, pp. 50–55, 2023. DOI: 10.36724/2072-8735-2023-17-4-50-55.
[62] C. Kirankumar. "Deploy a gRPC-based application on an Amazon EKS cluster and access it with an Application Load Balancer - AWS Prescriptive Guidance," [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/deploy-a-grpc-based-application-on-an-amazon-eks-cluster-and-access-it-with-an-application-load-balancer.html (visited on 11/30/2023).
[63] K. Indrasiri and D. Kuruppu, *gRPC: Up and Running*. O'Reilly Media, 2020, ISBN: 978-1-4920-5830-4.
[64] AWS. "User Guide - Amazon Simple Storage Service," [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide (visited on 10/25/2023).
[65] AWS. "User Guide - Amazon Relational Database Service," [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/ (visited on 10/25/2023).
[66] AWS. "Developer Guide - Amazon DynamoDB," [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide (visited on 10/25/2023).
[67] AWS. "Developer Guide - AWS Cloud Development Kit (AWS CDK) v2," [Online]. Available: https://docs.aws.amazon.com/cdk/v2/guide (visited on 10/25/2023).
[68] G. Brito, R. Terra, and M. T. Valente, *Monorepos: A Multivocal Literature Review*, 2018. arXiv: 1810.09477.
[69] Pantsbuild. "Welcome to Pants!" [Online]. Available: https://www.pantsbuild.org/ (visited on 10/25/2023).
[70] Apache Software Foundation. "Apache Parquet," [Online]. Available: https://parquet.apache.org/ (visited on 12/05/2023).
[71] AWS. "User Guide - Amazon Athena," [Online]. Available: https://docs.aws.amazon.com/athena/latest/ug/ (visited on 12/05/2023).
[72] IEEE and The Open Group, "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(TM))," *Base Specifications*, IEEE Std 1003.1TM-2017, no. 7, 2018. DOI: 10.1109/IEEESTD.2018.8277153.
[73] pandas. "API reference - pandas 2.1.3," [Online]. Available: https://pandas.pydata.org/docs/reference/index.html (visited on 12/05/2023).
[74] M. Cohn, *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004, ISBN: 978-0-321-20568-1.

[75] P. J. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4122.txt.
[76] ISO, *Date and Time - Representations for Information Interchange - Part 1: Basic Rules*, ISO 8601-1:2019/Amd 1:2022. 2022. [Online]. Available: https://www.iso.org/standard/81801.html.
[77] AWS. "Amazon API Gateway Features," [Online]. Available: https://aws.amazon.com/api-gateway/features/ (visited on 12/07/2023).
[78] AWS. "AWS Lambda," [Online]. Available: https://aws.amazon.com/lambda/ (visited on 12/07/2023).
[79] AWS. "Developer Guide - AWS Lambda," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg (visited on 12/26/2023).
[80] AWS. "Amazon EC2," [Online]. Available: https://aws.amazon.com/ec2/ (visited on 12/07/2023).
[81] AWS. "AWS Fargate," [Online]. Available: https://aws.amazon.com/fargate/ (visited on 12/07/2023).
[82] AWS. "Amazon DynamoDB," [Online]. Available: https://aws.amazon.com/dynamodb/ (visited on 12/13/2023).
[83] AWS. "Amazon DynamoDB pricing," [Online]. Available: https://aws.amazon.com/dynamodb/pricing/ (visited on 12/13/2023).
[84] AWS. "API Reference - Amazon Athena," [Online]. Available: https://docs.aws.amazon.com/athena/latest/APIReference/ (visited on 12/11/2023).
[85] AWS. "Developer Guide - Amazon Elastic Container Service," [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ (visited on 12/13/2023).
[86] AWS. "API Reference - Amazon RDS Data Service," [Online]. Available: https://docs.aws.amazon.com/rdsdataservice/latest/APIReference/ (visited on 12/08/2023).
[87] AWS. "Developer Guide - AWS Step Functions," [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/ (visited on 12/14/2023).
[88] AWS. "AWS Step Functions Pricing," [Online]. Available: https://aws.amazon.com/step-functions/pricing/ (visited on 12/14/2023).
[89] AWS. "AWS Step Functions FAQs," [Online]. Available: https://aws.amazon.com/step-functions/faqs/ (visited on 12/14/2023).
[90] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, 2007, ISBN: 978-0-596-52926-0.
[91] S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. O'Reilly Media, 2010, ISBN: 978-1-4493-8884-3.
[92] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013, ISBN: 978-1-4493-5806-8.
[93] R. T. Fielding, J. Gettys, J. C. Mogul, *et al.*, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt.
[94] Google. "Google JSON Style Guide," [Online]. Available: https://google.github.io/styleguide/jsoncstyleguide.xml (visited on 12/21/2023).

[95] A. Stuyvenberg. “The What, Why, and When of Mono-Lambda vs Single Function APIs.” (2021), [Online]. Available: https://dev.to/aws-builders/the-what-why-and-when-of-mono-lambda-vs-single-function-apis-5cig (visited on 10/04/2023).

[96] M. Ozkaya. “Decompose Lambda Functions — Monolithic, Microservices vs Single-Purposed Functions,” AWS Serverless Microservices with Patterns & Best Practices. (Apr. 9, 2023), [Online]. Available: https://medium.com/aws-serverless-microservices-with-patterns-best/decompose-lambda-functions-monolithic-microservices-vs-single-purposed-functions-f4b87465cb36 (visited on 12/19/2023).

[97] AWS. “REST API - Powertools for AWS Lambda (Python).” version 2.30.2, [Online]. Available: https://docs.powertools.aws.dev/lambda/python/latest/core/event_handler/api_gateway/ (visited on 12/23/2023).

[98] AWS. “The Lambda monolith,” [Online]. Available: https://serverlessland.com/content/service/lambda/guides/aws-lambda-operator-guide/monolith (visited on 12/19/2023).

[99] OpenAPI Initiative, *OpenAPI-Specification*, version 3.1.0, GitHub. [Online]. Available: https://github.com/OAI/OpenAPI-Specification/.

[100] TechEmpower. “Web Framework Benchmarks,” [Online]. Available: https://www.techempower.com/benchmarks (visited on 12/24/2023).

[101] SmartBear Software. “REST API Documentation Tool | Swagger UI,” [Online]. Available: https://swagger.io/tools/swagger-ui/ (visited on 12/25/2023).

[102] Amazon, *powertools-lambda-python*, version 2.30.2, GitHub, 2019. [Online]. Available: https://github.com/aws-powertools/powertools-lambda-python.

[103] Pallets, *Flask*, version 3.0.0, GitHub, 2010. [Online]. Available: https://github.com/slank/awsgi.

[104] M. Wedgwood, *awsgi*, version 0.2.7, GitHub, 2016. [Online]. Available: https://github.com/slank/awsgi.

[105] Django Software Foundation, *Django*, version 3.0.0, GitHub, 2005. [Online]. Available: https://github.com/django/django.

[106] J. Eremieff, *Mangum*, version 0.17.0, GitHub, 2021. [Online]. Available: https://github.com/jordaneremieff/mangum.

[107] S. Ramírez, *FastAPI*, version 0.106.0, GitHub, 2018. [Online]. Available: https://github.com/tiangolo/fastapi.

[108] Encode OSS Ltd., *Starlette*, version 0.34.0, GitHub, 2018. [Online]. Available: https://github.com/encode/starlette.

[109] Pydantic Services, *Pydantic*, version 2.5.3, GitHub, 2017. [Online]. Available: https://github.com/pydantic/pydantic.

[110] S. Ramírez. “FastAPI,” [Online]. Available: https://fastapi.tiangolo.com/ (visited on 12/25/2023).

[111] Litestar Org., *Litestar*, version 2.4.5, GitHub, 2021. [Online]. Available: https://github.com/litestar-org/litestar.

[112] Apache Software Foundation. “Apache Spark - A Unified engine for large-scale data analytics.” version 3.5.0, [Online]. Available: https://spark.apache.org/docs/latest/index.html (visited on 12/26/2023).

[113] V. Belov, A. Tatarintsev, and E. Nikulchev, "Choosing a Data Storage Format in the Apache Hadoop System Based on Experimental Evaluation Using Apache Spark," *Symmetry*, vol. 13, no. 195, 2021. DOI: 10.3390/sym13020195.

[114] AWS. "AWS CDK Python Reference," [Online]. Available: https://docs.aws.amazon.com/cdk/api/v2/python/ (visited on 12/29/2023).

[115] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995, ISBN: 0-201-63361-2.

[116] W. Badenhorst, *Practical Python Design Patterns: Pythonic Solutions to Common Problems*. Apress, 2017, ISBN: 978-1-4842-2679-7. DOI: 10.1007/978-1-4842-2680-3.

[117] Amazon, *aws-sdk-pandas*, version 3.4.2, GitHub, 2019. [Online]. Available: https://github.com/aws/aws-sdk-pandas.

[118] AWS. "Amazon ECR Public Gallery - AWS Lambda/python," [Online]. Available: https://gallery.ecr.aws/lambda/python (visited on 01/02/2024).

[119] AWS. "Choosing between REST APIs and HTTP APIs," [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html (visited on 01/02/2024).

[120] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003, ISBN: 978-0-321-12742-6.

[121] AWS, *boto3*, version 1.34.13, GitHub, 2013. [Online]. Available: https://github.com/boto/boto3.

[122] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, *pytest*, version 7.4.4, GitHub, 2004. [Online]. Available: https://github.com/pytest-dev/pytest.

[123] Pylint contributors, *Pylint*, version 3.0.3, GitHub, 1991. [Online]. Available: https://github.com/pylint-dev/pylint.

[124] J. Lehtosalo and contributors and Dropbox, *Mypy*, version 1.8.0, GitHub, 2012. [Online]. Available: https://github.com/python/mypy.

[125] K. Reitz, C. Benfield, and I. Cordasco, *requests*, version 2.31.0, GitHub, 2011. [Online]. Available: https://github.com/psf/requests.

[126] Apache Software Foundation, *Apache JMeter*, version 5.6.3, GitHub, 1998. [Online]. Available: https://github.com/apache/jmeter.

[127] AWS. "Operator Guide - AWS Lambda," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide (visited on 01/10/2024).

[128] AWS. "AWS Pricing Calculator," [Online]. Available: https://calculator.aws (visited on 01/11/2024).

[129] AWS. "API Reference - Amazon DynamoDB," [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference (visited on 01/11/2024).

List of Figures

3.1 Synchronous vs. asynchronous communication (adapted from [27]) . . . 8
3.2 Decoupled request invocation handling workflow using a queue with a mediator and multiple request workers (adapted from [38, p. 48]) 12
3.3 The ETL-inspired query processing stages in the context of the LonqAPI 13

4.1 The simplified architektur of the Data Layer at BHS Corrugated 19

5.1 Overview of the required LonqAPI architecture components and their interaction 24
5.2 The architecture of the Athena query-handling approch in combination with Lambda connectors 28
5.3 The architecture of the Fargate container cluster query-handling approach 29
5.4 The architecture of the Step Functions state machine query-handling approach 30
5.5 The LonqAPI architecture overview 32

7.1 The simplified LonqAPI implementation package structure 46
7.2 The abstract *BaseLonqStateMachine* construct and its configuration class . 48
7.3 The simplified process of implementing a new long-running query using *mdl-lonq* library constructs and utilities 49
7.4 The Athena POM query state machine shown in the AWS Step Functions editor 51
7.5 The implemented REST API calsses and request handlers, and their hierarchy in the layer architecture 53

8.1 The test client implementation flowchart diagram 57
8.2 Response times of the API for 10179 requests to the query resource endpoint in milliseconds 58

A.1 An example Athena POM query result 79
A.2 The process of implementing a new long-running query using *mdl-lonq* library constructs and utilities 91

List of Tables

6.1 Comparison of monolithic and single-purpose Lambda functions for API Gateway request handling . . . 38
6.2 Framework rating symbols . . . 42
6.3 REST API Python framework comparison . . . 42

8.1 The Athen query execution timestamps of the LonqAPI infrastructure relative to the POST request . . . 59
8.2 Exemplary monthly costs per AWS service, volume, and action for the LonqAPI infrastructure . . . 62
8.3 The classes and functions implemented to integrate DynamoDB into the LonqAPI . . . 63
8.4 Evaluation of the DynamoDB integration into the LonqAPI . . . 64

Appendix A

LonqAPI

Listing A.1: Athena POM Query Statement

```
SELECT area_id,
       pom_id,
       timestamp_epoch,
       timestamp_utc,
       coalesce(
               string_value,
               cast(boolean_value as varchar),
               cast(number_value as varchar)
       ) as value
FROM plant_pom
WHERE pom_id IN ( :pom_ids_param )
  AND to_unixtime(
    from_iso8601_timestamp(
    format('%s-%s-%sT%s:00', year, month, day, hour))
    ) BETWEEN :from_epoch_seconds
  AND :to_epoch_seconds
  AND timestamp_epoch >= :from_epoch
  AND timestamp_epoch <= :to_epoch
```

	area_id	pom_id	timestamp_epoch	timestamp_utc	value
0	123e4567-e89b-12d3-a456-426614174000	321e7654-e98b-21d3-b654-426614174123	1693389834796	2023-08-30 10:03:54.796	0.289894
1	123e4567-e89b-12d3-a456-426614174000	321e7654-e98b-21d3-b654-426614174123	1693809533716	2023-09-04 06:38:53.716	0.303648
2	123e4567-e89b-12d3-a456-426614174000	321e7654-e98b-21d3-b654-426614174123	1693809610630	2023-09-04 06:40:10.630	0.367085
3	123e4567-e89b-12d3-a456-426614174000	321e7654-e98b-21d3-b654-426614174123	1693389926568	2023-08-30 10:05:26.568	0.425844
4	123e4567-e89b-12d3-a456-426614174000	321e7654-e98b-21d3-b654-426614174123	1693389955441	2023-08-30 10:05:55.441	0.449185
5	123e4567-e89b-12d3-a456-426614174000	987e6543-e21b-32d1-cba9-426614174321	1693389986388	2023-08-30 10:06:26.388	918.450000
6	123e4567-e89b-12d3-a456-426614174000	987e6543-e21b-32d1-cba9-426614174321	1693390088501	2023-08-30 10:08:08.501	544.860000
7	123e4567-e89b-12d3-a456-426614174000	987e6543-e21b-32d1-cba9-426614174321	1693390131253	2023-08-30 10:08:51.253	595.180000
8	123e4567-e89b-12d3-a456-426614174000	987e6543-e21b-32d1-cba9-426614174321	1693809618356	2023-09-04 06:40:18.356	516.110000
9	123e4567-e89b-12d3-a456-426614174000	987e6543-e21b-32d1-cba9-426614174321	1693809619357	2023-09-04 06:40:19.357	531.880000

Figure A.1: An example Athena POM query result

Listing A.2: Data access function header to convert metric POM values to imperial units

```
def to_imperial(
    df: pd.DataFrame,
    col_name: str = 'value',
    transformed_col_name: str = 'value',
    pom_config_table_name: Optional[str] = None,
    session: Optional[boto3.Session] = None,
) -> None:
    '''
    Transform a column of a DataFrame from metric to imperial.

    Args:
        df (pd.DataFrame): The DataFrame to transform.
        pom_config_table_name (Optional[str]): The name of the
            DynamoDB table.
            If None, take parameter from aws parameter store.
            Defaults to None.
        col_name (str, optional): The column name. Defaults to
            'value'.
        transformed_col_name (str, optional):
            The name of the column containing the transformed
                value.
            Defaults to 'value'.
        session (Optional[boto3.Session], optional):
            The boto3 session to use. Defaults default session.
    '''
    ...
```

Listing A.3: Example JSON representation of a long-running query resource

```
{
  "clientId": "123e4567-e89b-12d3-a456-426614174000",
  "queryId": "321e7654-e98b-21d3-b654-426614174123",
  "status": "RUNNING",
  "resultUrls": [],
  "creationTime": "2024-01-01T12:30:00",
  "expirationTime": "2024-01-02T21:30:00",
  "createQueryUrl": "https://lonq-api-example.com/daas-history/
     long-running-queries/123e4567-e89b-12d3-a456-426614174000/
     process-data/areas/765e4321-e98b-21d3-b654-426614174123/
     point-of-measurement-query?startTime=1701385200000&endTime
     =1701400000000&pomIds=0e62e89a-9af7-4762-81f5-d6d2390edfff
     &imperial=False",
  "queryUrl": "https://lonq-api-example.com/daas-history/long-
     running-queries/123e4567-e89b-12d3-a456-426614174000/321
     e7654-e98b-21d3-b654-426614174123"
}
```

Listing A.4: Example idea of the usage of a REST API framework in an AWS Lambda function for the LonqAPI

```
def foo_handler(path_param, query_string_param, body):
    result = do_something(
        path_param,
        query_string_param,
        body
    )

    return {
        "status_code": 200,
        "body": result
    }

app = FrameworkApp()
app.add_route(
    "/foo/{path_param}",
    foo_handler,
    methods=["GET"]
)

def lambda_handler(event, context) -> dict:
    return app.resolve(event, context)
```

Listing A.5: Example Athena query response JSON of an UNLOAD POM data query


```
1  {
2    "QueryExecution": {
3      "EngineVersion": {
4        "EffectiveEngineVersion": "Athena engine version 3",
5        "SelectedEngineVersion": "Athena engine version 3"
6      },
7      "Query": "UNLOAD ( SELECT area_id,pom_id,timestamp_epoch,
          timestamp_utc, coalesce(string_value, cast(boolean_value
           as varchar), cast(number_value as varchar)) as value
          FROM plant_pom WHERE area_id = ? AND pom_id IN ( '
          b2dfc749-acd5-4c19-9766-af4a4d84b91f' ) AND to_unixtime(
           from_iso8601_timestamp( format('%s-%s-%sT%s:00', year,
          month, day, hour))) BETWEEN ? AND ? AND timestamp_epoch
          >= ? AND timestamp_epoch <= ? )TO 's3://example-mdl-daas
          -history-lonq-bucket/daasHistoryAthenaPomData/raw/822
          aa578-a77c-4992-9d6a-0b192fcbfdf2/' WITH (format='
          PARQUET', compression='None')",
8      "QueryExecutionContext": {
9        "Database": "machine_data_layer_dev"
10     },
11     "QueryExecutionId": "13b8359b-d260-46ef-9573-df82a1403fe9",
12     "ResultConfiguration": {
13       "EncryptionConfiguration": {
14         "EncryptionOption": "SSE_S3"
15       },
16       "OutputLocation": "s3://example-mdl-daas-history-lonq-
            bucket/daasHistoryAthenaPomData/raw/13b8359b-d260-46ef
            -9573-df82a1403fe9"
17     },
18     "ResultReuseConfiguration": {
19       "ResultReuseByAgeConfiguration": {
20         "Enabled": false
21       }
22     },
23     "StatementType": "DML",
24     "Statistics": {
25       "DataManifestLocation": "s3://example-mdl-daas-history-
            lonq-bucket/daasHistoryAthenaPomData/raw/13b8359b-d260
            -46ef-9573-df82a1403fe9-manifest.csv",
26       "DataScannedInBytes": 0,
27       "EngineExecutionTimeInMillis": 1134,
28       "QueryPlanningTimeInMillis": 551,
29       "QueryQueueTimeInMillis": 48,
30       "ResultReuseInformation": {
```

```
31          "ReusedPreviousResult": false
32        },
33        "ServicePreProcessingTimeInMillis": 79,
34        "ServiceProcessingTimeInMillis": 24,
35        "TotalExecutionTimeInMillis": 1285
36      },
37      "Status": {
38        "CompletionDateTime": 1703341557963,
39        "State": "SUCCEEDED",
40        "SubmissionDateTime": 1703341556678
41      },
42      "SubstatementType": "UNLOAD",
43      "WorkGroup": "daasHistoryAthenaPomDataLonqAthenaWorkGroup"
44    }
45  }
```

Listing A.6: The BaseLonqStateMachine and its configuration class

```
@dataclass
class BaseLonqStateMachineConfig:
    prefix_id: str
    bucket: aws_s3.IBucket


class BaseLonqStateMachine(constructs.Construct):
    '''
    Base class for lonq state machines.
    '''

    def __init__(
        self,
        scope: constructs.Construct,
        config: BaseLonqStateMachineConfig,
    ):
        super().__init__(scope, f'{config.prefix_id}
           LonqStateMachine')
        self.state_machine: Optional[sfn.StateMachine] = None
        self.config = config
        self.id = f'{config.prefix_id}LonqStateMachine'

    def build(self) -> sfn.StateMachine:
        query_chain = self._build_query_chain()

        first_state = query_chain.start_state
        if pre_query_chain := self._build_pre_query_chain():
            first_state = pre_query_chain.start_state
            pre_query_chain.next(query_chain)

        if post_query_chain := self._build_post_query_chain():
            query_chain.next(post_query_chain).next(
                sfn.Succeed(self, 'Success')
            )

        self.state_machine = sfn.StateMachine(
            self,
            self.id,
            state_machine_name=self.id,
            definition_body=sfn.DefinitionBody.from_chainable(
               first_state),
        )

        self._set_state_machine_permissions(self.state_machine)
```

```
43
44          return self.state_machine
45
46      @abstractmethod
47      def _build_query_chain(self) -> sfn.Chain:
48          pass
49
50      def _build_pre_query_chain(self) -> Optional[sfn.Chain]:
51          '''
52          Override this method to add a chain of states that
               should be executed before the query.
53          '''
54          return None
55
56      def _build_post_query_chain(self) -> Optional[sfn.Chain]:
57          '''
58          Override this method to add a chain of states that
               should be executed after the query.
59          '''
60          return None
61
62      def _set_state_machine_permissions(
63          self, state_machine: sfn.StateMachine
64      ) -> None:
65          self.config.bucket.grant_read_write(
66              state_machine,
67              f'{raw_result_bucket_prefix(self.config.prefix_id)
                   }/*',
68          )
69          if self.config.bucket.encryption_key:
70              self.config.bucket.encryption_key.
                   grant_encrypt_decrypt(
71                  state_machine,
72              )
```

Listing A.7: The POM state machine definition within the DaaS history API CDK stack

```
1  self.pom_state_machine = lonq_athena.AthenaLonqStateMachine(
2      self,
3      lonq_athena.AthenaLonqStateMachineConfig(
4          prefix_id='daasHistoryAthenaPomData',
5          bucket=lonq_bucket,
6          processor_lambda=query_processor_lambda,
7          readable_s3_bucket=persistence_stack.processed_poms.
              bucket,
8          database_name=persistence_stack.glue_database.
              database_name,
9          table_names=['plant_pom'],
10     ),
11 ).build()
```

Listing A.8: The simplified query processor Lambda handler creator function and its configuration classes

```
class BaseProcessorEvent(TypedDict):
    '''
    Base input to build a lonq processor.
    '''
    QueryId: str
    Processor: str
    ProcessorConfig: dict[str, Any]
    Result: dict[str, Any]


class LonqProcessorOutput(TypedDict):
    ResultBucket: str
    ResultKeys: list[str]


ProcessorCreator = Callable[
    [BaseProcessorEvent], query_processor.QueryProcessor[Any]
]


def create_handler(
    processor_creator: dict[str, ProcessorCreator]
) -> Callable[[dict[str, Any], LambdaContext],
   LonqProcessorOutput]:
    def handler(
        event: dict[str, Any], context: LambdaContext
    ) -> LonqProcessorOutput:

        # Check if event contains a valid lonq query output.
        # Extract a BaseProcessorEvent as processor_event.
        # Extract result bucket and prefix as lonq_context.
        # ...

        processor: query_processor.QueryProcessor[Any]
        processor = processor_creator[query['Processor']](
           processor_event)

        return LonqProcessorOutput(
            ResultBucket=lonq_context['ResultBucket'],
            ResultKeys=processor.process(
                bucket=lonq_context['ResultBucket'],
                prefix=lonq_context['ResultKeyPrefix'],
                query_id=query['QueryId'],
            ),
```

```
43          )
44
45      return handler
```

Listing A.9: The usage of the query processor Lambda handler creator function for the POM data imperial conversion processor

```
def build_pom_data_imperial_processor(
    event: query_processor_handler.BaseProcessorEvent,
) -> pom_data.PomDataImperialProcessor:
    '''
    Build a lonq query_processor that transforms pom data
       values to imperial units.
    '''
    processor_config = event['ProcessorConfig']
    if 'Bucket' not in processor_config:
        raise ValueError('Bucket not specified in processor 
           config')
    elif 'Keys' not in processor_config and 'Prefix' not in
       processor_config:
        raise ValueError('Keys or Prefix not specified in 
           processor config')

    return pom_data.PomDataImperialProcessor(
        extraction_iterator.S3ParquetIterator(
            session=session_manager.boto3_session,
            bucket=processor_config['Bucket'],
            keys=processor_config.get('Keys', None),
            prefix=processor_config.get('Prefix', None),
            chunked=processor_config.get('Chunked', False),
        ),
        session_manager,
        pom_config_table_name=os.environ['POM_CONFIG_TABLE'],
    )


handler = query_processor_handler.create_handler(
    processor_creator={
        'PomDataImperialProcessor':
           build_pom_data_imperial_processor,
    }
)
```

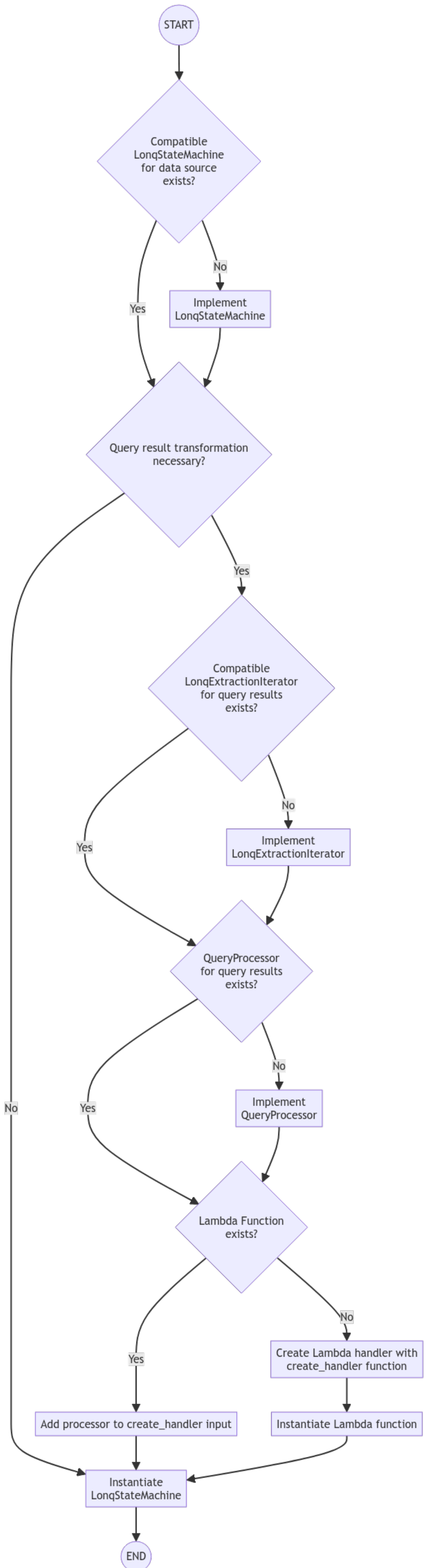

Figure A.2: The process of implementing a new long-running query using *mdl-lonq* library constructs and utilities

Listing A.10: The Litestar REST API request handler and router for creating a new POM data query

```
@litestar.post(
    '{client_id:uuid}/areas/{area_id:uuid}/point-of-measurement-values',
    sync_to_thread=False,
    summary='Create␣a␣query␣for␣point-of-measurement␣values␣in␣a␣given␣interval␣for␣a␣given␣area␣and␣pom␣ids.',
    status_code=status_codes.HTTP_202_ACCEPTED,
    return_dto=query.ExtendedLonqQueryDto,
    response_headers=[
        datastructures.ResponseHeader(
            name='Location',
            description='The␣location␣of␣the␣created␣query.',
            required=True,
            documentation_only=True,
        )
    ],
)
def create_pom_query(
    client_id: Annotated[uuid.UUID, parameter.ClientId],
    area_id: Annotated[uuid.UUID, parameter.AreaId],
    pom_ids: Annotated[list[uuid.UUID], parameter.PomIds],
    start_time_epoch: Annotated[int, parameter.StartTimeEpoch],
    end_time_epoch: Annotated[int, parameter.EndTimeEpoch],
    imperial: Annotated[
        bool,
        params.Parameter(
            bool,
            description='Whether␣to␣transform␣the␣values␣to␣imperial␣units.',
            title='Imperial',
            default=False,
        ),
    ],
    lonq_query_service: lonq_query_management.LonqQueryService,
    pom_data_query_service: pom_data_query.LonqPomDataQueryService,
    request: litestar.Request[Any, Any, Any],
) -> litestar.Response[query.ExtendedLonqQuery]:
    if start_time_epoch >= end_time_epoch:
        raise exceptions.ValidationException(
            detail='The␣start␣time␣must␣be␣before␣the␣end␣time.',
        )

```

```
40      query_id = uuid.uuid4()
41      execution_arn, output_type = pom_data_query_service.
            start_pom_query(
42          query_id=query_id,
43          area_id=area_id,
44          pom_ids=pom_ids,
45          start_time_epoch=start_time_epoch,
46          end_time_epoch=end_time_epoch,
47          imperial=imperial,
48      )
49
50      created_query = lonq_query_service.create_query(
51          query_id=query_id,
52          client_id=client_id,
53          execution_arn=execution_arn,
54          output_type=output_type,
55      )
56
57      return litestar.Response(
58          headers={'Location': query.get_query_path(client_id,
                query_id)},
59          content=query.create_extended_lonq_query(created_query,
                request),
60      )
61
62
63  pom_data_query_router = litestar.Router(
64      path='/long-running-queries/process-data/',
65      route_handlers=[create_pom_query],
66      tags=['Long␣Running␣Queries', 'Process␣Data', 'Point␣of␣
            Measurement'],
67  )
```

Listing A.11: The shortened manager class to initialize and hold sessions in AWS Lambda functions

```
class SessionManager:
    '''
    Wrapper class for managing instances of clients and
       sessions.
    Enables to avoid initializations for reused lambda
       environments, but for this the manger must be
       initialized at global scope of the lambda handler.
    To avoid unnecessary initializations, properties should be
       accessed only where needed.
    '''

    def __init__(self, boto3_session: Optional[boto3.Session] =
        None) -> None:
        self._initialized: dict[Any, object] = {}
        if boto3_session is not None:
            self._initialized[boto3.Session] = boto3_session

    @property
    def boto3_session(self) -> boto3.Session:
        return cast(
            boto3.Session,
            self._initialized.setdefault(
                boto3.Session,
                boto3._get_default_session(),
            ),
        )

    @property
    def s3_client(self) -> S3Client:
        return cast(
            S3Client,
            self._initialized.setdefault(
                S3Client,
                self.boto3_session.client('s3'),
            ),
        )

    .
    .
    .
```