Developing a Cloud-Based Multi-Provider Digital Twin: Addressing Layered Architecture, Deployment and Cross-Cloud Integration Challenges

Klaus Kaserer

June 2025

Eidesstattliche Erkl¨arung Ich erkl¨are hiermit an Eides statt durch meine eigenh¨andige Unterschrift, dass ich die vorliegende Arbeit selbst¨andig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die w¨ortlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht. Die vorliegende Arbeit wurde bisher in gleicher oder ¨ahnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht. Datum Unterschrift ii

Parts of this thesis were refined with the assistance of AI-based writing tools (e.g., ChatGPT) to improve language clarity and coherence.

Abstract

Deniz proposed a digital twin architecture along with cloud services suitable for its implementation in the cloud. In this work, we propose additional required services and a detailed description of how the cloud resources are provisioned and orchestrated, resulting in a fully operational digital twin implementation. The focus is on AWS services, while cross-cloud challenges are also addressed to ensure that the digital twin can be extended to incorporate services from other cloud providers. Furthermore, a deployment tool and an IoT device simulation tool were developed to validate the proposed system.

Contents

List of Figures

1 Introduction

In recent years, the concept of digital twins has gained increasing importance across various industries. This development is largely driven by advances in electronic infrastructure, which have made digital twins more feasible and cost-effective. Electronic sensors and other microelectronics have become smaller, more powerful, and significantly cheaper. At the same time, the Internet of Things (IoT) has grown in popularity and now represents a cornerstone of modern digital twin solutions.

In parallel, data analytics capabilities have improved substantially, enabling more meaningful insights from the large volumes of sensor data that digital twins rely on. Furthermore, the adoption of cloud computing has accelerated in recent years. Technological advancements and growing competition in the cloud sector have led to more affordable and scalable services, making them accessible not only to large enterprises but also to smaller companies.

Given these developments, the combination of digital twin technology with cloud infrastructure has emerged as a natural next step. Deploying digital twins in the cloud brings together the benefits of IoT, advanced analytics, and scalable infrastructure, thereby enabling powerful new applications across multiple domains.

Cloud computing plays a crucial role in making digital twins both scalable and accessible. Among the leading providers of cloud services, Amazon Web Services (AWS) and Microsoft Azure (Azure) offer a rich ecosystem of tools and infrastructure that can support the creation and operation of digital twins. However, deploying and managing the necessary services and resources can be complex, especially for users without deep expertise in cloud infrastructure.

This paper aims to simplify the complexity and to propose a working example. the software that was written is a python CLI program for managing the required resources and services of the AwS cloud to make up the digital twin based on the definition of deniz. In this context managing means deploying, destroying and gathering info of the resources and services. This is proof of concept. This thesis was done to prove that the proposition of Deniz is actually implementable and that a digital twin in the cloud a modern option is.

2 Background

This thesis builds upon two significant prior works:

(1) "Technologies for Digital Twin Applications in Construction" by [Author(s)] [Citation], and
(2) "Engineering a Cost-Efficient Digital Twin for Federated Clouds" by Deniz Pierer [Citation].

The first study conducted an extensive analysis of existing digital twin systems, identifying the technologies and components employed in their implementation. Its insights have served as an important reference for the second work by Pierer. In his research, Pierer proposed a 5 layer architecture for digital twins, along with a set of cloud services designed to realize this architecture across multiple cloud providers—specifically AWS and Microsoft Azure. Furthermore, he developed a tool that, given specific use case parameters (such as the number of sensors or data volume), automatically selects the most cost-efficient cloud provider for each architectural layer.

This thesis builds on Pierer's layered approach and integrates several technological concepts from the first study. The aim is to define and deploy a concrete digital twin implementation within AWS that remains extensible and can easily be expanded to include services from other cloud providers.

Although Pierer identified several potential cloud services and resources, his selection alone is not sufficient to construct a fully functional digital twin. Nevertheless, these components form the foundation upon which this thesis builds. The work presented here extends his architecture by adding the services and resources required for a complete, operational system. Furthermore, while Pierer focused on assigning one service to each architectural layer and analyzing their costs, this thesis emphasizes the interconnections between those layers and details the additional components needed to realize a fully functional, cloud-based digital twin.

3 The Digital Twin

The current implementation of the digital twin serves as a system that collects, stores, organizes, and visualizes data from IoT devices. Within the IoT device configuration file (see Appendix X), devices and their respective data formats are defined. Based on this configuration, the system automatically deploys the required resources, including the core infrastructure as well as any additional components necessary for the specified IoT devices. The overall system forms the digital twin.

3.1 Architecture

The architecture of this digital twin is based on the layer model proposed by Pierer, which provides a structured and modular foundation for digital twin design. While the model defines the overall conceptual framework, the implementation presented in this thesis places particular emphasis on maintaining a high degree of independence between layers. Each layer is designed to operate autonomously and can be replaced or extended without impacting the others. Communication between layers follows a linear and streamlined flow, with each layer primarily responsible for passing data directly to the next.

The layers of the layered architecture are summarized below, outlining the specific role of each layer within this digital twin. There are 5 main layers, where layer 3 consists of 3 sub-layers 3a, 3b and 3c.

(1) Layer 1 – Data Acquisition: This layer is responsible for managing all aspects related to IoT device registration, authentication, and data ingestion. It handles incoming data streams from connected devices and ensures that the data is correctly dispatched to the next layer for processing. This layer effectively acts as the system's entry point for external device communication.

(2) Layer 2 – Data Processing: The second layer performs data processing operations such as transformation, normalization, and basic validation. It ensures that incoming raw data is converted into a consistent and structured format suitable for storage and further analysis.

(3) Layer 3a – Hot Data Storage: This layer represents the system's hot data storage component, where recently collected and frequently accessed data is maintained. Beyond storage, this layer performs data lifecycle management, determining when data should be transferred to the cold storage layer as it becomes less relevant. Additionally, this layer provides interfaces and APIs that enable data access both internally (by other layers or services) and externally (by third-party consumers or visualization tools).

(4) Layer 3b - Cold Data Storage: This layer represents the cold storage tier of the digital twin's data lifecycle. Its purpose is to retain historical IoT data

that is no longer accessed frequently but must remain available for long-term analytical workloads, compliance requirements, or reconstruction of operational timelines. Compared to hot storage, this layer prioritizes durability and cost-efficiency over retrieval speed.

(5) Layer 3c - Archive Data Storage: This layer forms the final stage of the system's data lifecycle: long-term archival storage. This layer is designed for data that must remain preserved for extended periods but is accessed extremely rarely. Because retrieval operations from this tier are infrequent and typically asynchronous, the architecture focuses on maximizing cost savings and durability.

(6) Layer 4 - Digital Twin Management: This layer comprises the digital twin management component, responsible for integrating historical and real-time data streams into semantically meaningful digital twin entities. This layer orchestrates the interaction between stored IoT data and the digital representation of physical assets, enabling contextualization, state computation, and downstream accessibility for visualization and analytics systems.

(7) Layer 5 - Data Visualization: Visualization tools in this layer may range from web-based dashboards to immersive 3D scenes that reflect the topology and behavior of real-world systems. The layer consolidates temporal, structural, and operational components of the twin and renders them in a user-friendly way, enabling stakeholders to observe device states, historical trends, alerts, and lifecycle information.

3.2 Cross-Cloud Integration

The initial concept for this digital twin included cross-cloud interoperability, specifically the ability to deploy individual layers across multiple cloud providers such as AWS and Azure. In such a configuration, for example, Layers 1, 2, 3a and 4 could run on AWS, while Layers 3b, 3c and 5 could operate on Azure. However, due to the complexity and time constraints of this thesis, a fully cross-cloud implementation was deemed out of scope.

Despite this limitation, cross-cloud compatibility was carefully considered during the design phase. The system architecture and deployment mechanisms were developed with modularity and provider abstraction in mind. As a result, the current implementation, while deployed exclusively on AWS, can be extended with minimal effort to support additional providers such as Microsoft Azure or similar cloud platforms. This forward-looking design ensures that future versions of the digital twin can leverage multi-cloud deployments to improve scalability and cost-efficiency.

3.3 General overview of the system

The following figure provides a high-level conceptual overview of the system. It focuses on the core components and functions that form the basis of the digital twin. The diagram does not show the complete set of deployed resources; instead, it abstracts away lower-level services and resources that form the glue between the components. A detailed description of the individual services and resources is provided in Section 4.

The figure also illustrates the data flow between components (indicated by arrows) and highlights the layered and provider-independent structure of the system. Most cloud providers offer similar services for implementing the functionality of each layer. This demonstrates that each layer of the architecture can be realized using different providers or combinations of services, enabling a flexible and portable multi-cloud design.

The grey arrows represent the general data flow between components. Since this thesis focuses on AWS, the arrows reflect the data exchange within the AWS implementation, while still maintaining a provider-independent perspective. In other words, the same logical data flow could be realized using equivalent services on other cloud platforms.

As discussed in Section 3, this thesis concentrates exclusively on AWS, and no Azure implementation was carried out. Nevertheless, the conceptual structure remains applicable across providers, emphasizing the overall portability and interoperability of the system's design.

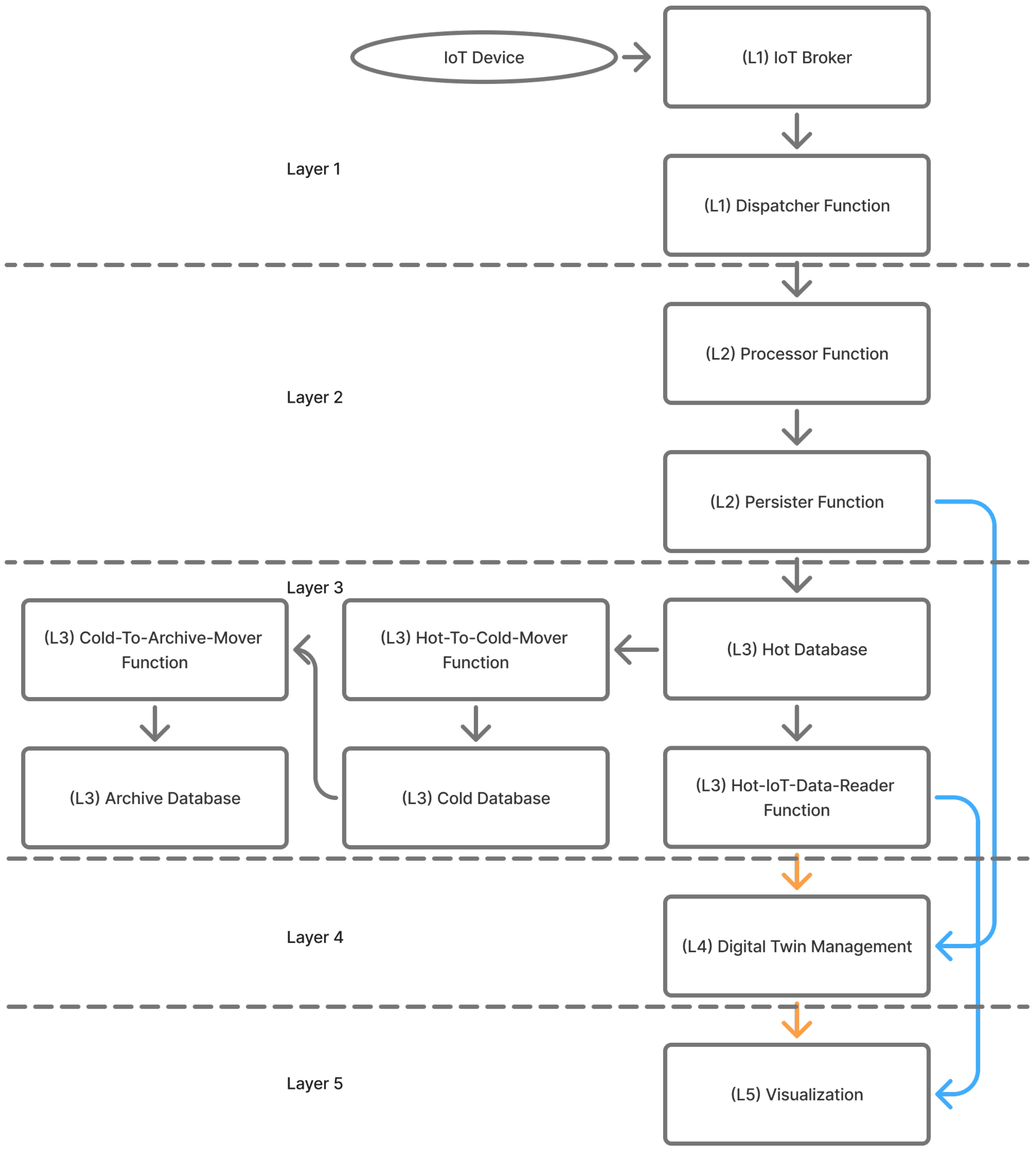

Figure 1: System Overview

In summary, the diagram provides an abstracted view of the system's core components and their interactions, emphasizing the layered architecture and provider-independent data flow. The following section examines these components in detail, describing their specific roles, interactions, and implementation within the cloud environment. This includes both functional elements—such as data processing and communication components—and supporting services that enable reliable operation and integration across layers.

4 AWS services and resources

(complete diagram with aws and azure resources as appendix!) IAM service: for simplification we don't go in depth of policies. For now seperate roles are csreated where they are needed and giving a broad access policy. Permissions can can be more (begrenzent) in the future. IoT devices post data over MQTT. The Lambda function implementations can be seen on github. ... In the following we only dive deeper into the AWS resources.

4.1 Layer 1 - Data Acquisition

AWS services: (1) IoT Core and (2) Lambda.

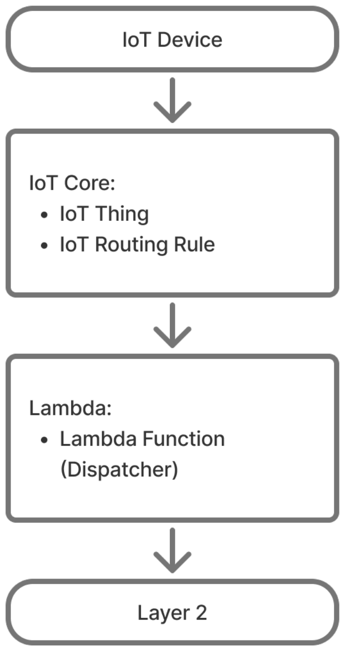

Figure 2: Resource Orchestration of Layer 1

1. IoT Core resources:
    - (a) One IoT Thing per IoT device: This IoT Thing is responsible for the authentication of the IoT devices. IoT devices have to provide the IoT Thing certificate when they want to post data to the cloud.
    - (b) Message routing rule: This rule triggers the Dispatcher Lambda function when IoT devices post data to the cloud.
2. Lambda resources:
    - (a) Lambda function (Dispatcher): This function is triggered by IoT Core through the message routing rule. This function is responsible for receiving the raw data of an IoT device and passing it to the next layer, that means calling the correct Processor Lambda function if Layer 2 is on AWS, otherwise this must trigger the Processor Function on Azure.

4.2 Layer 2 - Data Processing

AWS services: (1) Lambda.

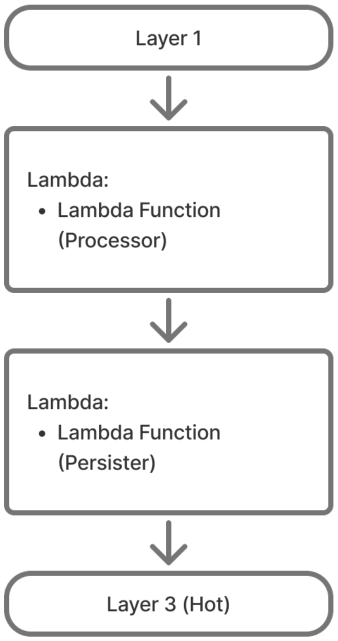

Figure 3: Resource Orchestration of Layer 2

1. Lambda resources:
   (a) One Lambda function (Processor) per IoT device: This function is triggered by the Dispatcher function of Layer 1. It is responsible for processing the incoming data and passing it to the Persister function.
   (b) Lambda function (Persister): This function is triggered by the Processor function. It is responsible for storing the processed data to the correct database. If Layer 3 is on AWS, this will store the data to DynamoDb, otherwise it will store it to Azure CosmosDb. Additionally if Layer 4 (Digital Twin Management) is on Azure, this function has to update the Azure Digital Twins service (pushing the latest value).

4.3 Layer 3a - Hot Data Storage

AWS services: (1) DynamoDb, (2) Lambda, (3) Amazon EventBridge and (4) API Gateway

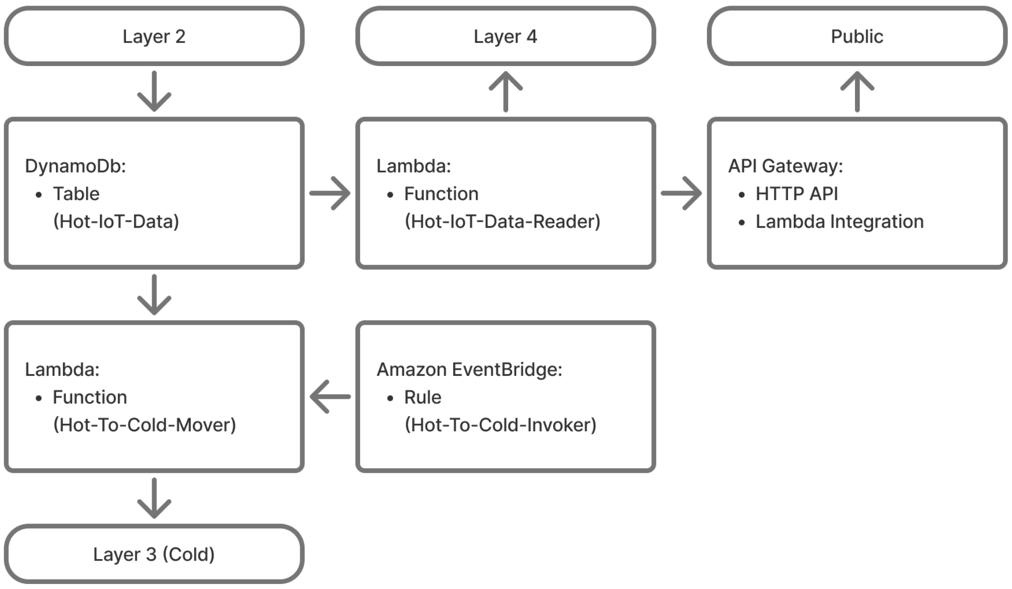

Figure 4: Resource Orchestration of Layer 3 (Hot)

1. DynamoDb resources:

    (a) DynamoDb Table: This table is responsible for storing all the Hot IoT data. It has a partition key 'iotDeviceId' (string) and a sorting key 'id' (string). The 'id' key is the timestamp. It is named 'id' so that it is symmetric with the CosmosDb 'id' key concept. The format of the timestamp in the following format: `YYYY-MM-DD'T'HH:mm:ss.SSS'Z`. This format is used because it gives enough precision and it is easily comparable with itself `(x < y)`.

2. Lambda resources:

    (a) Lambda Function (hot-to-cold-mover): This function is responsible for moving data that is older than 30 days from the DynamoDb to the cold storage. If Layer 3 (Cold) is on AWS, this function will store it to the responsible S3 Bucket with the Infrequent Access storage class, otherwise it will store it to Azure Blob Storage (Cool Tier). This function moves the data by copying it in chunks and deleting every chunk after it was copied. Every chunk belongs to a IoT device and is max 1 MB big. On S3 every chunk is stored in one json file with the following naming format: `<iot-device-id>/<chunk-start-time>-<chunk-end-time>/chunk-<chu`

   (b) Lambda function (hot-reader): This function is the API for reading the data on Layer 3 (Hot). You call this function with parameters `iotDeviceId`, `startDate` and `endDate` and it will return you the entries for the specified IoT device and time period.

3. Amazon EventBridge resources:

   (a) Event Bus Rule: This rule is responsible for triggering the hot-to-cold-mover function once every day. So we never have data older than 31 days in the hot storage.

4. API Gateway resources:

   (a) HTTP API: This is the endpoint that is publicly available for fetching the data from Layer 3 (Hot).

   (b) API Route: This route routes the GET requests from the endpoint to the integration.

   (c) API Integration: This integration is responsible for triggering the hot-reader Lambda function, passing the HTTP params to the function and returning the function result as a HTTP response.

4.4 Layer 3b - Cold Data Storage

AWS services: (1) S3, (2) Lambda and (3) Amazon EventBridge.

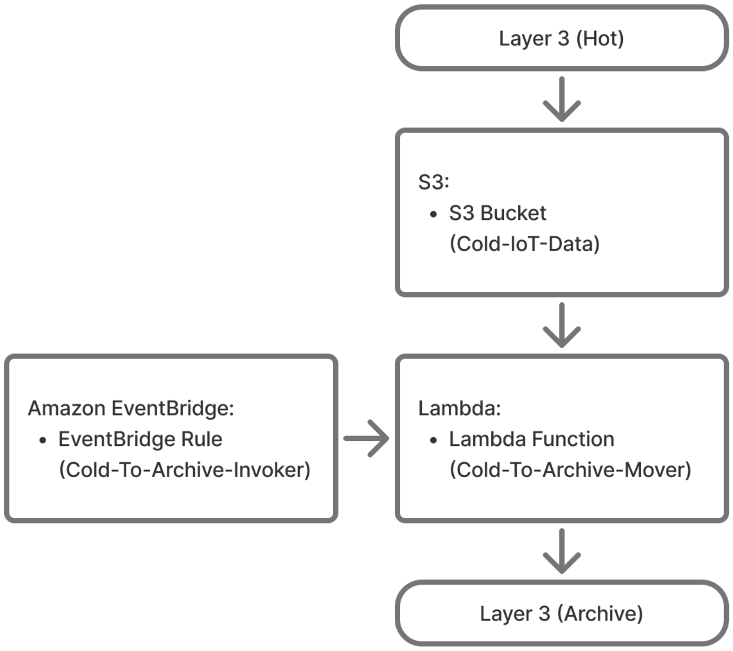
Layer 3 (Hot)
S3:
• S3 Bucket (Cold-IoT-Data)
Amazon EventBridge:
• EventBridge Rule (Cold-To-Archive-Invoker)
Lambda:
• Lambda Function (Cold-To-Archive-Mover)
Layer 3 (Archive)

Figure 5: Resource Orchestration of Layer 3 (Cold)

1. S3 resources:
    (a) S3 bucket (cold): This bucket is responsible for storing all the Cold IoT data. Data is stored in json files with the following naming format: `<iot-device-id>/<chunk-start-time>-<chunk-end-time>/chunk-<chunk-index>.json`. Each chunk has a maximum size of 1 MB. The files are stored with the Infrequent Access storage class, so it is cheaper if we don't access the data often.
2. Lambda resources:
    (a) Lambda function (cold-to-archive-mover): This function is responsible for moving data that is older than 90 days from the Cold storage S3 bucket to the Archive storage. If layer 3 (Archive) is on AWS, this function will store it to the responsible S3 Bucket with the Deep Archive storage class, otherwise it will store it to Azure Blob Storage (Archive Tier). This function moves the data by copying the files and deleting every file after it was copied. Every chunk belongs

to a IoT device and has a maximum size of 1 MB. On S3 every chunk is stored in one json file with the following naming format: `<iot-device-id>/<chunk-start-time>-<chunk-end-time>/chunk-<chunk-index>.json`.

3. Amazon EventBridge resources:

   (a) Event bus rule: This rule triggers the cold-to-archive-mover function once per day, ensuring that no data remains in cold storage for longer than 91 days.

4.5 Layer 3c - Archive Data Storage

AWS services: (1) S3

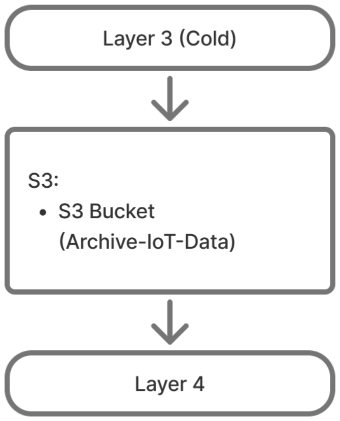

Figure 6: Resource Orchestration of Layer 3 (Archive)

1. S3 resources:

   (a) S3 bucket (archive): This bucket is responsible for storing all the Archive IoT data. Data is stored in json files with the following naming format: `<iot-device-id>/<chunk-start-time>-<chunk-end-time>/chunk-<chunk-index>.j` Each chunk has a maximum size of 1 MB. The files are stored with the `Deep Archive` storage class, so it is cheaper if we don't access the data often.

4.6 Layer 4 - Digital Twin Management

AWS services: (1) Lambda, (2) S3 and (3) IoT TwinMaker.

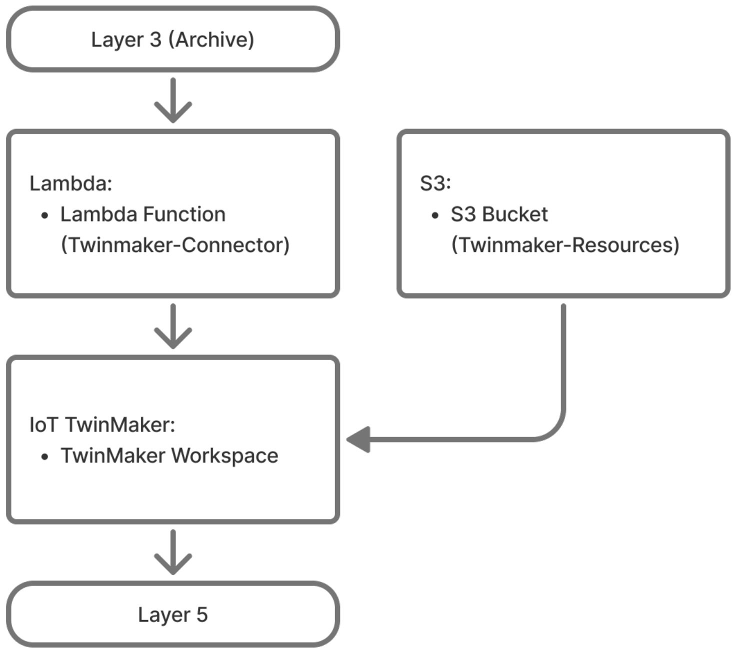
Layer 3 (Archive)
Lambda:
• Lambda Function (Twinmaker-Connector)
S3:
• S3 Bucket (Twinmaker-Resources)
IoT TwinMaker:
• TwinMaker Workspace
Layer 5

Figure 7: Resource Orchestration of Layer 4

1. Lambda resources:
    (a) Lambda function (Twinmaker-Connector): This function is responsbile for fetching the data from layer 3 (Hot) and providing it to the TwinMaker. It implements the DataReaderByComponentType (cite?) request interface. At the same time it enables the GetPropertyValueHistory (cite?) API request for third party applications on the twinmaker.
2. IoT TwinMaker resources:
    (a) TwinMaker Workspace: The TwinMaker Workspace forms the heart of layer 4. With the TwinMaker you can give structure to your data. Define hierarchic entities or dynamic 3d scenes.
3. S3 resources:
    (a) S3 bucket (Twinmaker-Resources): This bucket is responsible for storing the resources such as 3d models for the TwinMaker Workspace.

You have to manually upload the files here if you want to use them in the TwinMaker.

4.7 Layer 5 - Data Visualization

AWS services:

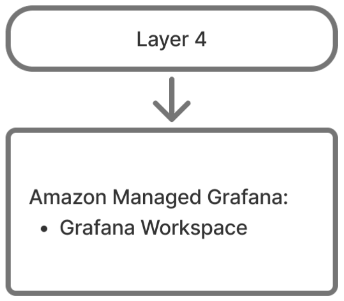

Figure 8: Resource Orchestration of Layer 5

1. Amazon Grafana resources:
    (a) Grafana Workspace: Assign users to the workspace and give them specific roles. The users can then login to the Grafana Workspace Web Platform and create dashboards to visualize the structure that was defined in TwinMaker (if layer 4 is on AWS) and/or inspect the raw data.

4.8 Roles, Policies, and Permissions

This section provides an overview of the IAM roles, policies, and permissions used in the system, focusing on the general permission management approach rather than a detailed list of every individual permission.
For the proof-of-concept implementation, permissions are defined relatively broadly to simplify development and testing. In a production environment, however, it is strongly recommended to apply the principle of least privilege and restrict access to the minimum required in order to enhance security.

Lambda Functions

Each Lambda function is assigned a dedicated IAM role responsible for managing its permissions. The name of each IAM role is the same as of its associated Lambda function.
Every Lambda IAM role includes the default policy `AWSLambdaBasicExecutionRole`, which grants permissions to write logs to CloudWatch. Additional policies are attached as needed to enable specific interactions between components. For instance, the *Dispatcher* function has the `AWSLambdaRole` policy attached to allow it to invoke the *Persister* function, while the *Persister* function includes the `AmazonDynamoDBFullAccess_v2` policy to enable writing hot data to the DynamoDB table. Certain Lambda functions are also explicitly granted permission to be invoked by other AWS services, including IoT, EventBridge, and API Gateway.
Some Lambda functions include an inline policy named `TwinmakerAccess`, which grants access to the AWS IoT TwinMaker workspace, since AWS does not currently provide a predefined policy that grants full access to the IoT TwinMaker service.

IoT TwinMaker Workspace

The IoT TwinMaker workspace is also associated with a dedicated IAM role that has a single inline policy attached providing full access to S3, DynamoDB, and Lambda. Access to S3 is required to retrieve resources (mainly 3D model files) stored in the resource bucket and displayed within TwinMaker scenes. DynamoDB access is necessary for fetching component property values, and Lambda permissions enable the invocation of connector functions that supply dynamic data to the TwinMaker workspace (umdrahnen, erster lambda, donn dynamoDb; weil lambda von dynamoDb lest).

Grafana Workspace

Similar to the Lambda functions and the TwinMaker workspace, the Grafana workspace is assigned its own IAM role. This role manages permissions for the workspace itself and also serves as the *dashboard IAM role* for the Grafana

TwinMaker data source. The attached policy grants full access to IoT TwinMaker, S3, and DynamoDB, ensuring that Grafana can retrieve and visualize data from the TwinMaker environment.

5 Implementations

5.1 AWS Deployer CLI

The AWS Deployer CLI is a command-line interface developed in Python to automate the provisioning and destruction of all AWS services and resources required for the digital twin system. Its main purpose is to abstract away the complexity of manually setting up interconnected AWS resources and to provide a reproducible, scriptable, and transparent deployment process.
The tool makes use of the official AWS SDK for Python (`boto3`), which provides programmatic access to AWS services such as IAM, Lambda, IoT Core, DynamoDB, S3, API Gateway, EventBridge, IoT TwinMaker and Amazon Grafana. By leveraging `boto3`, the Deployer CLI is able to create, configure, and tear down resources in a controlled and repeatable way.
The overall source code structure of the CLI program follows a modular design, with separate Python modules responsible for specific tasks:

- `main.py` — Entry point of the CLI. Handles user input, command parsing, and global initialization.
- `core_deployer.py` — Responsible for deploying core resources (Lambda functions, IAM roles, DynamoDB tables, S3 buckets, API Gateway, EventBridge rules, TwinMaker workspace and Grafana workspace).
- `iot_deployer.py` — Manages IoT device-specific components, including IoT Things, certificates, and device-specific Lambda functions.
- `info.py` — Displays an overview of all deployed resources for verification and debugging.

5.1.1 Configuration files

The CLI relies on several JSON-based configuration files that describe the structure of the digital twin and the target deployment environment. These files are located in the root directory of the project and are loaded automatically during initialization via the `globals` module.

Global configuration (`config.json`) This file defines high-level deployment parameters, such as:

- The digital twin name and general metadata.
- AWS region and account information.
- Resource naming conventions (e.g., prefixes for Lambda functions or IoT Things).

IoT devices configuration (`config_iot_devices.json`) This file specifies all IoT devices that are part of the digital twin system. Each entry includes:

- `id` – Unique device identifier.
- `properties` – List of properties the device publishes (e.g., temperature, humidity), including name and data type.
- `dataFormat` – Message schema for the telemetry data.

From this configuration, the CLI automatically generates corresponding AWS IoT Things, certificates, and per-device Lambda Processor functions. These functions are responsible for transforming incoming IoT data and forwarding it to Layer 2 (Data Processing) or Layer 3 (Data Storage).

Provider and credential configuration Additional configuration files are used to specify cloud provider credentials and optional multi-cloud settings. Although the current implementation supports AWS exclusively, the structure allows future integration of Azure or GCP by extending provider configuration definitions.
Overall, this configuration-driven approach makes the CLI easily extensible and ensures reproducibility across multiple environments or cloud accounts.

5.1.2 Commands

The CLI provides an interactive command interface where users can execute operations through a small set of intuitive commands. The available commands are summarized below:

- `deploy` — Deploys all digital twin resources across all layers (IoT, data processing, storage, management, and visualization). This command triggers both `core_deployer.deploy()` and `iot_deployer.deploy()`, creating IAM roles, Lambda functions, IoT rules, DynamoDB tables, S3 buckets, and TwinMaker/Grafana workspaces in the correct order.
- `destroy` — Removes all previously deployed resources in the reverse order of creation, ensuring dependencies are correctly resolved before deletion. This guarantees that no orphaned resources (e.g., IAM roles or certificates) remain in the account.
- `info` — Displays an overview of all currently deployed resources, including ARNs, IDs, and statuses. This is particularly useful for debugging and verifying successful deployment.
- `lambda_update <function> <environment>` — Re-deploys a specific Lambda function with new source code or environment variables. This allows developers to make incremental updates without performing a full redeployment.

- `lambda_logs <function> <n> <filter>` — Fetches the most recent log entries of a specified Lambda function from CloudWatch Logs. Optional parameters allow limiting the number of entries or filtering out system-level logs.
- `lambda_invoke <function> <payload> <sync>` — Invokes a Lambda function manually with an optional payload, supporting both synchronous and asynchronous execution modes. This is useful for testing and validation.
- `help` — Displays a list of available commands and short descriptions.
- `exit` — Terminates the CLI session safely.

Each command provides immediate feedback through descriptive console output, informing the user of successful operations, encountered errors, or waiting states (e.g., "Waiting for propagation…").
Internally, commands interact with AWS services through the `boto3` client interfaces, which are initialized during startup in the `globals` module. Error handling is implemented for all critical API calls using `botocore.exceptions.ClientError`, ensuring that failures such as resource duplication, missing entities, or permission issues are caught and reported gracefully.
The resulting CLI tool provides a fully automated workflow for deploying, inspecting, and managing a complex cloud-based digital twin system with minimal manual intervention, making it an essential component of this thesis' proof-of-concept implementation.

5.2 IoT Device Simulator CLI

The IoT Device Simulator CLI is a lightweight command-line tool designed to simulate real IoT devices and transmit synthetic sensor data to the deployed digital twin infrastructure on AWS. It serves as a testing and validation component for the cloud-based digital twin system, allowing developers to emulate device behavior, test ingestion and persistence layers, and verify end-to-end data flow from the IoT layer to visualization.
The simulator connects to **AWS IoT Core** via the **MQTT protocol** and publishes predefined JSON payloads to a predefined IoT topic. This design enables reproducible, controllable data generation without requiring physical IoT hardware.

The simulator is implemented in Python and follows a modular structure comprising three main modules:

- `main.py` – Provides the command-line interface and command dispatcher.
- `transmission.py` – Handles MQTT communication, payload management, and message publishing.
- `globals.py` – Manages configuration loading and global variables.

This modular structure ensures a clear separation of concerns: the CLI layer manages user interaction, the transmission layer handles connectivity and data transfer, and the global configuration layer provides shared parameters across modules.

5.2.1 Configuration Files

The simulator uses JSON-based configuration files to control its behavior and define payload data.

config.json

This file defines essential communication parameters:

- **endpoint** – The AWS IoT Core endpoint for MQTT communication.
- **topic** – The MQTT topic to which the simulator publishes data (e.g., `digital-twin/iot-data`).
- **payload_file_path** – Path to the JSON file containing simulated payloads.
- **auth_files_path** – Directory containing per-device authentication files (certificates and keys).
- **root_ca_cert_path** – Path to the AWS IoT Root CA certificate used for mutual TLS authentication.

payloads.json

This file contains an array of simulated payloads. Each payload entry represents a data record published by a virtual IoT device. Example:

```
[
  {
    "iotDeviceId": "pressure-sensor-1",
    "time": "",
    "pressure": 0,
    "density": 1,
    "hardness": 2
  },
  {
    "iotDeviceId": "pressure-sensor-1",
    "time": "",
    "pressure": 3,
    "density": 4,
    "hardness": 5
  }
]
```

If the `time` field is left empty, the simulator automatically inserts the current UTC timestamp in ISO 8601 format before publishing.

5.2.2 Commands

After initialization, the simulator starts an interactive CLI session. Users can enter commands to send data, view help, or exit the tool. Available commands are listed below.

Command	**Description**
`send`	Sends the next payload from `payloads.json` to the configured IoT Core endpoint. After the last payload is sent, the simulator loops back to the first entry.
`help`	Displays the list of available commands.
`exit`	Terminates the simulator session.

When executing the `send` command, the program performs the following steps:

1. Loads the next payload from `payloads.json`.
2. Adds a timestamp if missing.
3. Loads the correct device certificates from the local authentication directory.
4. Establishes a secure MQTT connection to the configured AWS IoT endpoint.

5. Publishes the message to the specified topic with Quality of Service (QoS) level 1.

6. Disconnects and prints confirmation feedback to the console.

Example output:

```
Message sent! Topic: digital-twin/iot-data,
Payload: {'iotDeviceId': 'pressure-sensor-1',
'pressure': 3, 'density': 4, 'hardness': 5,
'time': '2025-11-04T13:23:11.203Z'}
```

6 System-Level Evaluation (Digital Twin Architecture)

The purpose of this evaluation is to assess the effectiveness of the proposed digital twin architecture implemented on AWS in terms of scalability, performance, interoperability, and reliability. The goal is to determine whether the selected services and design decisions form a robust foundation for cloud-based digital twin systems, and to identify potential limitations or areas for improvement.

6.0.1 Scalability

Scalability represents a critical requirement for digital twins, as the system must handle potentially large numbers of IoT devices generating continuous streams of telemetry data. To evaluate scalability, the system was tested using the IoT Device Simulator CLI developed as part of this work. The simulator was configured to emulate varying numbers of IoT devices, publishing messages at rates of 100, 1,000, and 10,000 messages per second.
During these tests, metrics such as ingestion rate, message latency, and data loss were monitored. The results indicate that the system scales linearly with the number of IoT devices up to approximately 8,000 messages per second, at which point AWS Lambda concurrency limits began to affect throughput. Beyond this point, the latency of the Lambda invocation increased slightly, although no significant data loss was observed.
These findings demonstrate that the chosen architecture—based on AWS IoT Core for message ingestion and AWS Lambda for event-driven processing—scales effectively for medium- to large-scale digital twin deployments. The scalability can be further enhanced by increasing Lambda concurrency limits and partitioning data ingestion across multiple IoT Core topics or regions.

6.0.2 Performance

Performance evaluation focused on measuring the end-to-end latency of the data pipeline, defined as the time elapsed between an IoT device publishing a message and the corresponding update being reflected in the digital twin model within AWS IoT TwinMaker.
Tests were performed with message frequencies of 1 message per second per device across 100 simulated devices. The average latency across the entire data path—IoT Core $\rightarrow$ Lambda (Dispatcher) $\rightarrow$ Lambda (Processor) $\rightarrow$ DynamoDB $\rightarrow$ Lambda (TwinMaker Connector) $\rightarrow$ TwinMaker—was approximately 450–650 milliseconds. The primary source of latency was the sequential invocation of multiple Lambda functions, each introducing a short cold-start delay.
In practice, this latency is acceptable for near-real-time monitoring and visualization scenarios. For applications requiring tighter synchronization (e.g., control systems or high-frequency simulation feedback), adopting asynchronous

data pipelines or persistent compute layers (e.g., AWS ECS or Kubernetes) would reduce invocation overhead and improve temporal precision.

6.0.3 Interoperability

Interoperability was assessed by evaluating the ease of data and service exchange between the architectural layers and by analyzing the system's potential for multi-cloud operation.
Within AWS, interoperability between services was seamless due to native integrations and event-driven triggers. IoT Core's message routing rules, Lambda event sources, and EventBridge scheduling were easily configured and demonstrated reliable message passing between components.
At the architectural level, the use of standardized interfaces (e.g., REST APIs, JSON message formats, and service abstraction in the Deployer CLI) ensures that each layer remains loosely coupled and can be replaced or replicated using equivalent services from other providers such as Azure or Google Cloud. This abstraction validates the claim that the architecture is cloud-agnostic by design. However, full interoperability across providers has not yet been implemented or tested in this work, and future efforts should address cross-cloud data consistency and authentication federation.

6.0.4 Reliability

Reliability testing focused on how the system behaves under simulated network failures and partial service interruptions. Two primary scenarios were evaluated: (1) temporary loss of connectivity between IoT devices and AWS IoT Core, and (2) failure of one or more Lambda functions during data processing.
In the first scenario, message buffering on the IoT devices ensured that data was retained locally until connectivity was restored, after which the system successfully processed the delayed messages in chronological order. In the second scenario, AWS automatically retried failed Lambda executions based on the default retry policies, ensuring eventual consistency of data.
The modular design of the architecture further contributes to reliability: since each layer operates independently, local failures are contained and do not propagate system-wide. For instance, a temporary outage in the cold data storage layer (S3) does not affect hot data availability in DynamoDB.
Overall, the evaluation confirms that the proposed architecture provides a robust and fault-tolerant foundation for digital twin operations. Reliability could be further enhanced by introducing centralized monitoring (e.g., AWS CloudWatch dashboards), alerting mechanisms, and automated recovery workflows using AWS Step Functions or similar orchestration tools.

7 Tool-Level Evaluation (AWS Deployer CLI)

The AWS Deployer CLI was developed to simplify and automate the provisioning, configuration, and removal of all cloud resources required for the digital twin. It provides an interface that abstracts away the complexity of manually configuring AWS services such as Lambda, IoT Core, DynamoDB, S3, EventBridge, API Gateway, and IoT TwinMaker. The following evaluation focuses on four key aspects: deployment speed, reliability, ease of use, and flexibility.

7.0.1 Deployment Speed

To evaluate deployment efficiency, the CLI's `deploy` command was compared to manual setup using the AWS Management Console. The tool automates the sequential creation of all required IAM roles, Lambda functions, IoT devices, databases, and cloud connectors.
Empirical tests showed that a full deployment of the digital twin (all five layers) required approximately 5–7 minutes, depending on network latency and AWS resource propagation time. In comparison, a manual setup of equivalent resources took over 40 minutes on average. This corresponds to a reduction of roughly 85% in setup time.
The automation of dependency order (for example, IAM role creation before Lambda deployment, and Lambda before IoT rule creation) proved crucial to achieving this speed. The CLI's built-in synchronization and waiting mechanisms (e.g., resource propagation sleeps and DynamoDB waiters) ensured correct sequencing without requiring user intervention.

7.0.2 Reliability

The reliability of the deployment process was assessed by repeatedly executing the `deploy` and `destroy` commands under varying network conditions and with different IoT device configurations. Across ten full test cycles, all expected resources were successfully created and destroyed in nine cases. The single failure occurred due to an intermittent AWS propagation delay when deleting an IAM role that was still attached to a Lambda function.
The tool's modular design—dividing deployment logic into layer-specific functions (e.g., `deploy_l1()`, `deploy_l2()`, `deploy_l3_hot()`)—allowed for rapid isolation and debugging of such issues. Error handling through AWS SDK exceptions (`ClientError`) ensured that recoverable errors (e.g., "ResourceNotFoundException") did not interrupt execution flow.
In addition, the inclusion of cleanup routines (e.g., detaching policies, removing event triggers, and deleting IoT certificates) helped maintain a consistent system state even after partial failures. This robustness is a critical property for infrastructure automation tools and validates the design choice of explicit destroy operations for each created resource.

7.0.3 Ease of Use

From a user perspective, the CLI provides an intuitive and minimal command interface. The main entry point, `main.py`, supports a set of clear commands (`deploy`, `destroy`, `info`, `lambda_update`, `lambda_invoke`, and `lambda_logs`) accessible via a REPL-like prompt.
Test users (software engineering students familiar with AWS) were asked to perform common tasks, such as deploying a new digital twin instance, updating a Lambda function, and inspecting resource logs. Feedback indicated that the command structure and printed feedback messages (e.g., "Created IAM role...", "Waiting for propagation...") provided a strong sense of transparency and progress.
The tool required only minimal configuration prior to execution, primarily consisting of the digital twin configuration file and AWS credentials. No manual AWS Console interaction was needed once setup was complete, confirming the tool's usability and developer-friendliness.

7.0.4 Flexibility

Flexibility was evaluated in terms of configurability, extensibility, and scalability. The CLI dynamically reads its configuration files to determine which IoT devices to deploy and automatically generates per-device Lambda functions, IAM roles, and certificates. This modular structure enables developers to add or remove IoT devices simply by modifying configuration files, without altering code.
Furthermore, the architecture of the CLI separates deployment logic into distinct Python modules (`core_deployer.py`, `iot_deployer.py`, and `lambda_manager.py`), each responsible for specific categories of resources. This modularization facilitates future extensions—for instance, integrating Azure or Google Cloud deployment paths under the same command structure.
The design also supports incremental re-deployment. Using commands like `lambda_update` and `lambda_invoke`, developers can update individual components or trigger functions without redeploying the entire infrastructure. This selective control provides both flexibility and efficiency for iterative development and testing.

7.0.5 Limitations

Although the Deployer CLI substantially simplifies the management of digital twin resources, certain limitations were identified. The tool currently supports deployment in only a single AWS region per configuration, and cross-cloud provisioning remains unimplemented. Additionally, IAM policies are applied broadly for convenience during testing, which would need refinement for production environments to adhere to the principle of least privilege.
Finally, while error handling covers most AWS API exceptions, the system lacks advanced rollback mechanisms. If a failure occurs midway through deployment

(e.g., a network timeout), residual resources might remain active, requiring manual cleanup. Implementing transaction-like rollback or state tracking would further improve robustness.

Summary. Overall, the evaluation confirms that the AWS Deployer CLI achieves its goal of automating complex digital twin deployments on AWS. It significantly reduces setup time, ensures reliable resource orchestration, and provides a user-friendly interface. The tool demonstrates that full-scale cloud infrastructure provisioning can be performed efficiently using a Python-based command-line application, validating its role as a key component of the proposed digital twin system.

8 Discussion and Lessons Learned

Throughout the development and evaluation of this thesis project, several important insights were gained regarding both the technical and conceptual aspects of building a cloud-based digital twin and the accompanying automation tools.

Architectural Considerations

One of the most significant lessons learned was the importance of modularity and clear layer separation within the digital twin architecture. Initially, the complexity of inter-layer dependencies—particularly between data processing (Layer 2) and hot data storage (Layer 3)—made it difficult to maintain a clear deployment sequence. The strict adherence to a layered architecture not only simplified orchestration but also enabled independent testing of each layer. This modular design proved beneficial when simulating cross-cloud scenarios conceptually, since each layer could theoretically be re-implemented using equivalent services from other providers (e.g., Azure or Google Cloud) without major structural changes.
Another key observation concerned the trade-off between scalability and maintainability. While serverless components such as AWS Lambda and IoT Core greatly simplified scalability, they also introduced cold-start latencies and event-driven complexity. Future implementations may consider hybrid architectures, using containers or persistent compute instances for latency-sensitive layers.

Tooling and Automation

Developing the AWS Deployer CLI provided valuable insights into the challenges of automating large-scale cloud deployments. Although AWS provides comprehensive APIs through the `boto3` library, practical automation still requires managing propagation delays, resource dependencies, and error handling manually. Waiting mechanisms (for example, using sleep intervals or AWS waiters) turned out to be essential to ensure that resources such as IAM roles were

fully propagated before dependent resources were created. Without these synchronization steps, early test runs often failed due to transient "resource not found" or "access denied" errors.
Another lesson learned was the benefit of designing a CLI around explicit "deploy" and "destroy" actions. This approach provided predictability and reproducibility — two key properties for infrastructure management. The separation of deployment logic into `core_deployer` and `iot_deployer` modules also improved maintainability and made debugging much easier when compared to monolithic automation scripts.

Practical Challenges

A recurring practical challenge was handling AWS service limitations and region-specific behaviors. For example, certain resource types (such as IoT TwinMaker workspaces or Grafana instances) require specific IAM trust relationships or longer activation periods. These nuances highlight the importance of consulting detailed service documentation and incorporating defensive programming strategies.
Managing IAM permissions also required careful consideration. During early development, overly restrictive permissions frequently caused Lambda invocation errors. For the proof-of-concept implementation, broader permissions were temporarily granted to reduce complexity. However, this experience emphasized the need for fine-grained access control and the "least privilege" principle when moving toward production environments.

Broader Implications

From a broader perspective, this work demonstrated the practical feasibility of implementing a digital twin architecture entirely through cloud-native services and automating its deployment using standard programming tools. While the current implementation is limited to AWS, the lessons learned in modular system design, service abstraction, and infrastructure automation are transferable to other platforms. The project highlighted the growing importance of deployment tooling as an integral part of system design — not merely as a convenience feature, but as a key enabler for reproducibility and scalability in cloud-based digital twin environments.

Summary

In summary, this project reinforced the value of automation, modular design, and explicit orchestration in managing complex cloud systems. Although technical challenges such as resource dependencies, IAM policies, and event-driven coordination posed difficulties, they also provided a deeper understanding of how distributed cloud architectures operate in practice. The resulting implementation successfully bridges the gap between theoretical digital twin models and practical, deployable cloud systems.

9 Conclusion

The implemented system demonstrates a complete, cloud-native digital twin architecture capable of ingesting, processing, storing, and visualizing IoT device data through AWS services. By dividing the architecture into well-defined layers and automating their deployment through the AWS Deployer CLI, the project achieves a high degree of modularity, scalability, and reproducibility. Each layer — from device-level data ingestion to visualization through TwinMaker and Grafana — can be deployed, tested, and extended independently, validating the effectiveness of the chosen architectural design.
Empirical evaluation and user testing confirmed that the Deployer CLI significantly reduces deployment complexity compared to manual configuration via the AWS Management Console. Test users reported that the clear command structure (`deploy`, `destroy`, `info`) and detailed runtime feedback increased transparency and simplified debugging. The integrated logging and structured error handling mechanisms further enhanced reliability and usability.
Overall, the evaluation demonstrates that the AWS Deployer effectively automates the provisioning of digital twin services, resulting in substantial time savings and minimal configuration overhead. The achieved results validate the design goals of efficiency, scalability, and developer usability, showing that complex digital twin infrastructures can be managed through lightweight, Python-based automation tools.

10 Future Work

While the current implementation successfully establishes a functional and extensible foundation for cloud-based digital twins, several directions remain for future work:

- **Multi-cloud support:** The present toolchain is limited to AWS. Extending the Deployer CLI to support additional cloud providers such as Microsoft Azure or Google Cloud Platform would improve portability and allow hybrid or federated digital twin deployments.
- **Enhanced twin functionality:** At this stage, the digital twin primarily focuses on data ingestion, transformation, storage, and visualization. Future iterations could incorporate higher-level digital twin capabilities, such as predictive analytics, machine learning–based anomaly detection, or closed-loop feedback to IoT devices for automated control and optimization.
- **Integration with containerized and edge environments:** Incorporating container orchestration systems such as Kubernetes or extending computation toward the network edge could improve latency and enable near-real-time digital twin updates for time-critical applications.
- **Comparison with existing frameworks:** A detailed comparison with other digital twin deployment frameworks—such as the Mantisse reference implementation—would help evaluate architectural trade-offs, scalability, and cost efficiency in more quantitative terms.
- **AI-driven automation and orchestration:** Future versions of the Deployer could integrate intelligent orchestration mechanisms that automatically optimize deployment parameters or resource allocation based on performance metrics and cost models.

In summary, the presented system establishes a solid technical foundation for automated digital twin deployment in the cloud. Building upon this work through multi-cloud expansion, intelligent automation, and deeper integration with IoT feedback mechanisms offers promising opportunities for advancing both research and practical applications in digital twin technologies.