Bachelor’s Thesis

Information and Communications Technology

2025

Miika Ravi

Evaluating the Use of Rust in AWS Lambda

– Performance, Cost and Development Efficiency

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2025 | 57 pages

Miika Ravi

Evaluating the Use of Rust in AWS Lambda

- Performance, Cost and Development Efficiency

Serverless computing services, like AWS Lambda, enable applications to automatically scale and execute functions in response to different events. One way to optimize Lambda functions is by choosing a high-performance programming language, which can improve runtime speed and reduce cold start latency – potentially leading to lower operational costs.

This thesis analyzes the use of the Rust programming language in AWS Lambda functions within the SADE Booster IoT Asset system and evaluates its potential benefits and drawbacks. The aim was to evaluate how rewriting the most performance-critical Node.js-based Lambda functions with Rust would affect performance, costs, and development efficiency.

The test results showed that Rust-based Lambda functions offered significant performance improvements, especially in cold start times and memory usage. This led to lower operational costs, especially when using optimized memory configurations. Rust can therefore be a worthwhile choice for performance-critical Lambda functions, despite the additional development time and effort.

Keywords:

Rust, Cloud Computing, Software Development, Amazon Web Services, Performance Optimization

Opinnäytetyö (AMK) | tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2025 | 57 sivua

Miika Ravi

Rust-ohjelmointikielen käytön evaluointi AWS Lambda -ympäristössä

- Suorituskyky, kustannukset ja kehitystehokkuus

Serverless-laskentapalvelut, kuten AWS Lambda, mahdollistavat sovellusten automaattisen skaalaamisen ja funktioiden suorittamisen vastauksena erilaisiin tapahtumiin. Yksi tapa optimoida Lambda-funktioita on valitsemalla suorituskykyinen ohjelmointikieli, mikä voi parantaa suoritusaikaa ja vähentää kylmäkäynnistysviivettä – näin käyttökustannukset voivat myös pienentyä.

Tämä opinnäytetyö analysoi Rust-ohjelmointikielen käyttöä Lambda-funktioissa SADE Booster IoT Asset -järjestelmässä ja arvioi sen mahdollisia etuja sekä haittoja. Tavoitteena oli arvioida, miten suorituskykykriittisten Node.js-pohjaisten Lambda-funktioiden uudelleenkirjoittaminen Rustilla vaikuttaisi suorituskykyyn, käyttökustannuksiin ja kehitystehokkuuteen.

Testitulokset osoittivat, että Rust-pohjaiset Lambda-funktiot paransivat suorituskykyä merkittävästi, erityisesti kylmäkäynnistyksissä ja muistinkäytössä. Tämä johti alhaisempiin käyttökustannuksiin, etenkin optimoiduilla muistiasetuksilla. Rust voi siis olla kannattava valinta suorituskykykriittisiin Lambda-funktioihin, vaikka kehitys vaatiikin enemmän aikaa ja resursseja.

Asiasanat:

Rust, Pilvilaskenta, Ohjelmistokehitys, Amazon Web Services, Suorituskyvyn optimointi

Contents

List of abbreviations 7

1 Introduction 8

2 SADE Booster IoT Asset 10

2.1 IoT Hardware Solution 10

2.2 Web & Mobile Application 11

2.3 Cloud Service 11

3 Amazon Web Services 13

3.1 AWS Lambda 13

3.2 AWS AppSync 14

3.3 Amazon DynamoDB 17

3.4 AWS IoT Core 18

3.5 Amazon Cognito 20

4 Rust Programming Language 22

4.1 Rust features 22

4.1.1 Mutability 22

4.1.2 Type system 24

4.1.3 Ownership and borrowing 27

4.1.4 Cargo 29

4.2 Rust for AWS Lambda 30

4.3 Challenges and trade-offs 32

5 Current Implementation 33

5.1 devicesStatesGet 33

5.2 usersGet 34

5.3 organizationsGet 37

6 New Implementation 39

6.1 Codebase structure 39

6.2 Implementation of Rust-based Lambda functions 40

6.3 Development efficiency 41

7 Testing 43

7.1 Performance 43

7.2 Costs 47

7.3 Optimal memory configuration 51

8 Conclusion 53

References 55

Figures

Figure 1. The building blocks of SADE Booster IoT Asset. [5] 10

Figure 2. Lambda execution environment lifecycle. [7] 14

Figure 3. Lambda cold starts and latency. [7] 14

Figure 4. Example GraphQL Schema. 15

Figure 5. AWS AppSync flow. [12] 16

Figure 6. DynamoDB's supported data types. 18

Figure 7. Example AWS IoT Core thing. 19

Figure 8. Example of a mutable variable in Rust. 23

Figure 9. Example of a constant declaration in Rust. 23

Figure 10. Example of shadowing in Rust. 24

Figure 11. Example of Rust's integer type annotation. 25

Figure 12. Example of Rust's floating type number annotation. 26

Figure 13. Example of Rust's Boolean type annotation. 26

Figure 14. Example of Rust's character type annotation. 26

Figure 15. Example of Rust's tuple type annotation. 27

Figure 16. Example of Rust's array type annotation. 27

Figure 17. Example of Rust's ownership rules. 28

Figure 18. Example of an immutable reference in Rust. 29

Figure 19. Example of a mutable reference in Rust. 29

Figure 20. "Hello" Lambda function in Rust. 31

Figure 21. Sequence diagram of the devicesStatesGet Lambda function. 34
Figure 22. Sequence diagram of the usersGet Lambda function. 36
Figure 23. Sequence diagram of the organizationsGet Lambda function. 38

Tables

Table 1. Built-in Integer Types in Rust. 25
Table 2. Lambda performance comparison with 1024 MB memory amount. 44
Table 3. Rust-based Lambda performance comparison with different memory amounts. 46
Table 4. Lambdas’ billed durations and estimated costs comparison with 1024 MB memory amount. 48
Table 5. Rust-based Lambdas’ billed durations and estimated costs comparison with different memory amounts. 50
Table 6. Optimal Rust-based Lambdas’ memory configurations between performance and cost. 51

List of abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWS	Amazon Web Services
CLI	Command Line Interface
IAM	Identity and Access Management
IoT	Internet of Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
R&D	Research and Development
SDK	Software Development Kit

1 Introduction

Serverless computing is an application development and execution model that enables the development and deployment of applications on cloud-managed infrastructure, where the cloud provider manages provisioning, scaling, and resource maintenance. In 2014, Amazon introduced AWS Lambda, the first serverless platform [1]. AWS Lambda is able to run code in response to different events, automatically scaling to match the demand. This is ideal for applications that have fluctuating workloads.

AWS Lambda does have some performance limitations – such as cold start latency, where the function experiences a delay when invoked after a period of inactivity. The issue becomes especially critical in applications, where quick responses are important. High cold start latency in performance-critical applications can lead to negative user experience, depending on the functionality of the Lambda.

A factor that influences the performance of Lambdas is the choice of a programming language. Node.js, a popular choice, is widely used in Lambda functions. However, it comes with garbage collection overhead which can hinder performance. In contrast, Rust offers high performance, memory safety, and low runtime overhead, making it a compelling choice for optimizing Lambda functions.

Previous studies on the performance of AWS Lambda functions have shown that the choice of programming language significantly impacts execution speed and cold start latency. According to research comparing the performances of various languages [2-4] have indicated that interpreted languages tend to have higher cold start delays and longer execution compared to compiled languages like Rust. Based on these findings, it is reasonable to expect that Rust could offer measurable performance improvements over the current Node.js implementation.

This thesis evaluates the impact of rewriting the most performance-critical Lambda functions from Node.js to Rust within SADE Innovations' Booster IoT Asset. SADE Booster IoT Asset is a collection of hardware, embedded software, connectivity, and cloud services with mobile and web applications. It provides a ready-made foundation for custom IoT solutions. By reimplementing Booster's selected Lambda functions with Rust, the aim is to evaluate whether Rust can enhance the Lambdas' performance and lower AWS costs, while also assessing the impact on development efficiency. The focus will be on the following aspects:

- Measuring Lambdas' performance differences between Node.js and Rust
- Evaluating AWS cost implications
- Assessing the development effort and trade-offs associated with the Rust rewriting process

These findings will provide insights for SADE Innovations to determine whether Rust is a viable alternative for writing Lambda functions in Booster, and whether a larger-scale migration would be beneficial.

The next section focuses on the system the Rust-based Lambdas will be implemented in.

2 SADE Booster IoT Asset

SADE Booster IoT Asset (Booster) is a modular and scalable hardware and software platform that provides an end-to-end solution for IoT systems. It is made up of AWS cloud infrastructure, various IoT platforms, and both web and mobile applications, as depicted in Figure 1.

Figure 1. The building blocks of SADE Booster IoT Asset. [5]

The core concept of Booster is to accelerate the R&D process through its building blocks. Rather than starting a project from scratch, Booster offers a ready-made platform for integrating connected devices. The following sections examine these building blocks in more detail.

2.1 IoT Hardware Solution

Booster includes a range of modular IoT hardware components for different use cases and requirements. These solutions provide full cloud connectivity and support different deployment scenarios, from ultra-low-power wireless sensors to edge computing solutions.

IoT CoreNode is the foundation for connected devices, providing secure cloud connectivity via WiFi, cellular, or Ethernet. It supports both battery-powered and on-premise installations and can be equipped with an optional display, touch

interface, GPS, and various sensors. CoreNode devices integrate with AWS IoT Core, enabling remote device management and over-the-air (OTA) updates. [6]

The IoT EdgeNode enables devices to respond to real-time events, operate offline, and receive updates from the cloud. It runs a Linux-based OS with AWS Greengrass, making it suitable for low-latency, edge computing tasks that require local processing. [6]

IoT Wireless Sensor Node is designed for ultra-low-power applications, enabling long-term battery operation. It uses Bluetooth, Bluetooth Mesh, and Thread for local connectivity. [6]

2.2 Web & Mobile Application

The web application offers an interface for visualizing device data and managing IoT systems remotely. It includes user authentication, role-based access control, and predefined UI components for login and registration. Users can monitor their device fleet, analyze historical data, and configure devices. The web app is built with React. [5]

The mobile application extends Booster’s functionality to mobile devices. While capable of monitoring multiple devices, its design is optimized for single-device control. It is built with React Native for both Android and iOS. [5]

In the next section and for the rest of the thesis, the focus will be on the cloud side of Booster.

2.3 Cloud Service

The Booster cloud service is designed with a fully serverless and event-driven architecture, eliminating the need for infrastructure management while enabling automatic scaling, high availability, and cost-efficient pay-per-use pricing.

It is composed of nine separately deployed services, each responsible for implementing a set of use cases. Devices interact with the cloud via an MQTT-

based interface, while client applications communicate through GraphQL APIs. The Lambda functions to be rewritten with Rust are a part of the “devices” and “users” services.

Each of Booster’s services communicate through well-defined interfaces. Services interact primarily via event-driven mechanisms where AWS Lambda functions are triggered by specific events, such as state changes or new data being received.

While services handle their own data, they expose controlled access to this data through their individual GraphQL APIs. Real-time updates are delivered to clients through GraphQL subscriptions, eliminating the need for client-side polling.

The Booster cloud service uses AWS Cognito User Pools for authentication, and AWS IAM roles for authorization. Each service’s access is controlled via IAM roles, granting users permission to interact with GraphQL APIs. Custom JavaScript resolvers handle the additional authorization checks based on user roles and features stored in JSON Web Signature tokens.

The next section will go through the AWS services that are relevant to this thesis.

3 Amazon Web Services

The Lambda functions this thesis focuses on are responsible for querying and processing data related to users, devices, and organizations. The key AWS services involved are AWS Lambda, AWS AppSync, Amazon DynamoDB, AWS IoT Core, and AWS Cognito.

AWS Lambda executes serverless functions triggered by GraphQL API calls, while AppSync acts as the GraphQL gateway, routing requests from client applications. DynamoDB stores and is used to fetch device states and organizational data. IoT Core is used for fetching device-related data, and Cognito manages and is used to fetch user data. These components handle and deliver Booster's real-time data.

3.1 AWS Lambda

AWS Lambda is a serverless computing service that enables code execution without the need to manage servers. Developers write code and deploy it to AWS Lambda, which is triggered by events from AWS services, incoming messages, or scheduled times.

AWS Lambda uses a pay-per-use model, where costs are incurred only during execution. AWS automatically handles scaling based on demand, ensuring that if there is a high volume of requests, Lambda will scale by adding instances, maintaining performance without the need to manage infrastructure. Each Lambda function is configurable, with settings such as allocated memory, maximum execution time, and concurrency limits.

A Lambda function is defined by a handler, which contains the logic to be executed when triggered. The handler expects two inputs: a payload (typically a JSON message containing data) and context (metadata about the call). When receiving the trigger, the handler initializes the function, processes the payload and executes the function's logic. After the function's logic is executed, Lambda terminates its runtime and extensions. This lifecycle is represented in Figure 2.

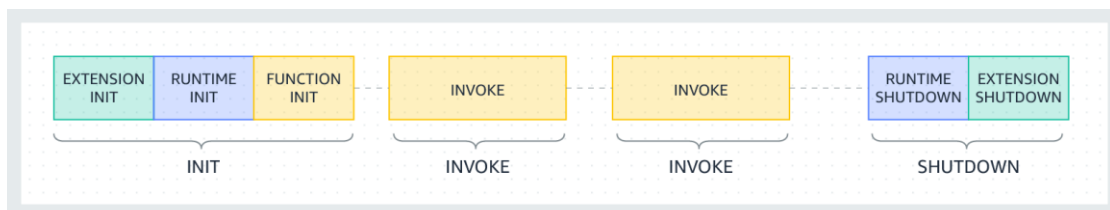

Figure 2. Lambda execution environment lifecycle. [7]

Previously, cold starts were briefly mentioned as part of the initialization phase of Lambda functions. During this phase, Lambda downloads the function code, initializes the execution environment, and runs any code outside of the main handler. [7] Only after this, Lambda runs the actual handler code. This is represented in Figure 3.

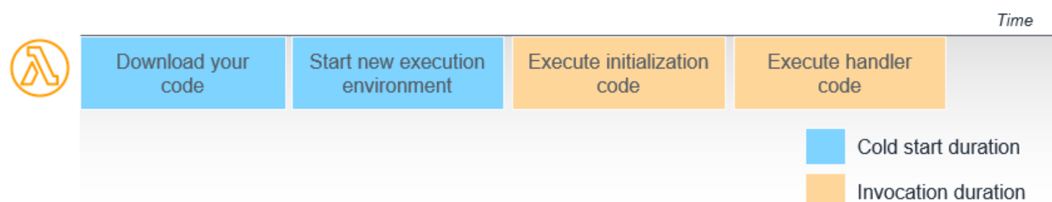

Figure 3. Lambda cold starts and latency. [7]

3.2 AWS AppSync

AWS AppSync provides secure, serverless, and high-performance GraphQL and Publish/Subscribe APIs to connect applications and services to data and events. [8]

GraphQL is a query and manipulation language for APIs that allows precise querying and returns reliable results. [9]

The GraphQL schema acts as the foundation and blueprint of a GraphQL API, defining the shape of its data. It also specifies how the data will be handled. GraphQL schemas are written in Schema Definition Language (SDL), which includes types and fields arranged in a set structure:

- Types: Types define the shape and behavior of the data. GraphQL supports many types, with the most common being objects and scalars.
- Fields: Fields exist within types and hold the values requested from the GraphQL service. The defined fields determine how the data is structured in queries and responses. [10]

An example GraphQL schema could look similar to this (Figure 4):

```
type User {
  id: ID!
  username: String!
  email: String!
}
type Query {
  users: [User]
}
type Mutation {
  createUser(id: ID!, username: String!, email: String!): User
  deleteUser(id: ID!): Boolean
}
```

Figure 4. Example GraphQL Schema.

As seen in Figure 4, the schema defines three types. The User type is an object holding three fields: id, username, and email. The Query type is responsible for retrieving data from the source, while the Mutation type is used to modify data.

AWS AppSync supports different data sources such as AWS Lambda, Amazon DynamoDB, Amazon OpenSearch, Amazon EventBridge, HTTP endpoints, and relational databases. When a client sends a request, it goes through a single schema endpoint. The schema then retrieves, processes, and sends the data back to the client. [11] This is demonstrated in Figure 5.

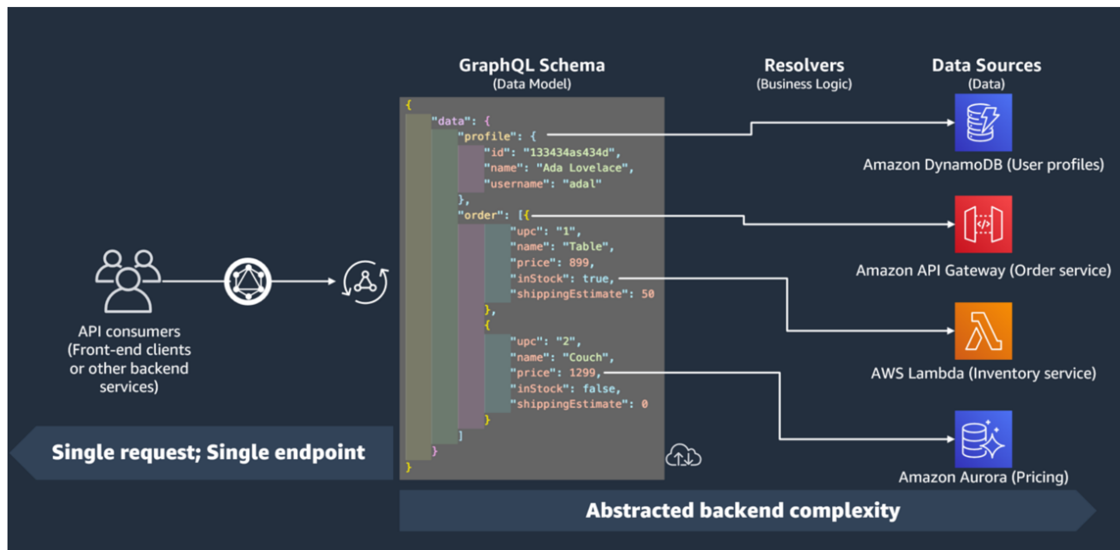

Figure 5. AWS AppSync flow. [12]

As seen in Figure 5, after a request is received, a resolver runs its code to process data from the appropriate data source. A resolver is a piece of code responsible for determining how handle the data for a specific field when the service receives a request. Resolvers are most commonly used handle state-changing operations for queries, mutations, and subscriptions. The resolver processes the client’s request and returns the result. [12]

There are two types of resolvers: unit resolvers and pipeline resolvers.

A unit resolver consists of code defining a request handler and a response handler that interact with a data source. The request handler returns the request payload used to call the data source, while the response handler takes the data source’s response and transforms it into a GraphQL response to resolve the field. [12]

Similar to a unit resolver, a pipeline resolver has request and response handlers, but it also includes a series of functions executed sequentially. They are used for complex operations, such as authorization checks before fetching data. Unlike unit resolvers, they are not tied to a specific data source and can interact with multiple sources. Functions in a pipeline are reusable.

3.3 Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless, NoSQL database service. DynamoDB is designed to handle large payloads, and like AWS Lambda, it scales automatically. Its core components are tables, items, and attributes.

- Tables: Store data, similar to other databases.
- Items: Data records within tables, uniquely identified by a primary key.
- Attributes: Data elements attached to items, comparable to fields or columns in other databases.

DynamoDB supports two types of primary keys:

- Partition key: A single attribute that determines the item’s storage location through a hash function.
- Composite primary key: Combines a partition key and sort key, making it possible for multiple items to share the same partition key while having different sort keys.

To support more flexible queries, DynamoDB allows the creation of secondary indexes. Secondary indexes are data structures that include a subset of attributes from a table and provide an alternate key to support queries. DynamoDB supports two types of secondary indexes:

- Global secondary index: Uses a partition key and sort key that can be different from the base table’s keys, allowing cross-partition queries.
- Local secondary index: Uses the same partition key as the base table, but has a different sort key. [13]

DynamoDB supports different data types, which are categorized into three groups. Figure 6 illustrates these data types with their descriptors.

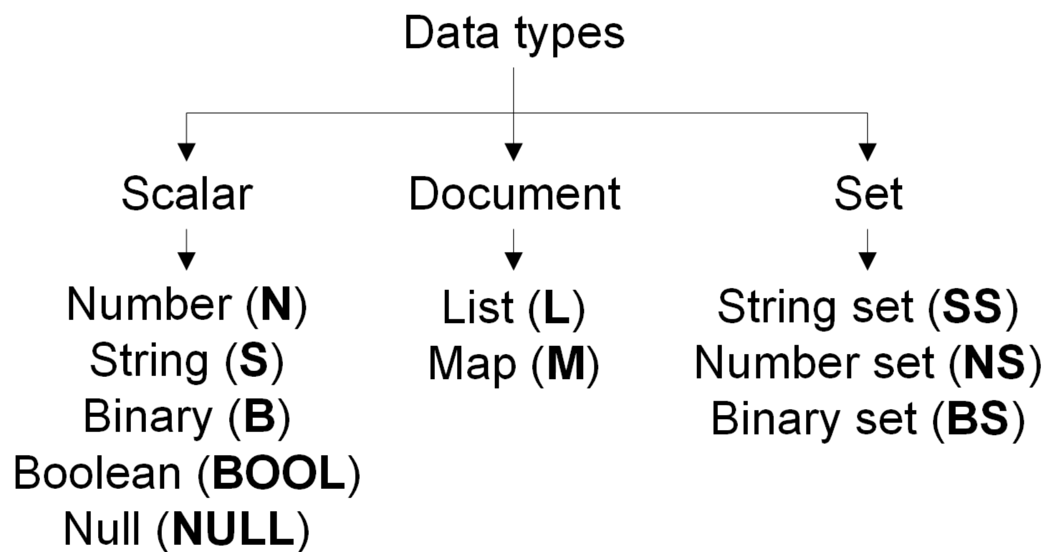

Figure 6. DynamoDB's supported data types.

DynamoDB stores data in partitions. Each partition allocates storage for a table and is backed by solid state drives. Partitions are automatically replicated across multiple Availability Zones within an AWS Region. [14]

There are two throughput modes for managing the capacity of a table, on-demand mode, and provisioned mode. In on-demand mode, the table automatically scales its throughput capacity based on actual usage, meaning charges are based on the throughput that is used. Provisioned mode requires specifying the number of reads and writes per second that the table should handle. Charges are based on the hourly provisioned read and write capacity. The tables interacted with in this thesis are on-demand.

3.4 AWS IoT Core

AWS IoT Core is a managed cloud platform that enables connected devices to interact with cloud applications and other devices easily and securely. It has support for billions of devices and trillions of messages, processing and routing the messages to AWS endpoints and other devices. [15]

AWS IoT Core provides a message broker that facilitates the communication between devices and IoT Core. It supports devices and clients that use MQTT and MQTT over WebSocket Secure (WSS) protocols for publishing and subscribing to messages, as well as devices and clients using HTTP to publish messages. [16]

AWS IoT Core includes a device registry to manage IoT devices, also known as things. Things can be grouped by their type, making it easier to manage and organize devices in the registry. Information about things is stored in the registry as JSON data. Example of a thing is depicted in Figure 7.

```json
{
  "thingName": "LivingRoomLightSwitch",
  "thingTypeName": "LightSwitch",
  "attributes": {
    "status": "off",
    "model": "SmartSwitch"
  },
  "version": 2
}
```

Figure 7. Example AWS IoT Core thing.

Things also have a “shadow” property. Shadows allow a device’s state to be accessible to applications and other services. They act as a reliable data store that enables devices, apps, and other cloud services to share data and remain connected or disconnected without losing the device’s state.

Each shadow has a reserved MQTT topic and HTTP URL that support the get, update, and delete actions. JSON shadow documents are used to store and retrieve data. These documents contain a state property with the following fields:

- desired: the desired states of device properties.
- reported: the current state of the device.
- delta: the difference between desired and reported states. [17]

Each connected device or client requires credentials to interact with AWS IoT. All communication with AWS IoT is secured using Transport Layer Security (TLS) to protect the data between AWS services. AWS IoT Core authenticates devices using X.509 certificates or Amazon Cognito identities, ensuring only authorized devices can access the platform. [18]

3.5 Amazon Cognito

Amazon Cognito is a cloud-based identity service for web and mobile applications. It consists of user directory management, an authentication server, and an authorization service for OAuth 2.0 access tokens and AWS credentials. [19]

Cognito has two key components, user pools and identity pools.

- User pools: User directories, that enable the sign-up and sign-in functionality for applications. User pools handle tasks like user registration, authentication, account recovery, and multi-factor authentication. User pools support customizable authentication flows, which can be used to authorize access to different resources.
- Identity pools: Directories of federated identities, that enable the creation of unique identities and assigning user permissions. Identity pools generate temporary AWS credentials for application users. Identity pools support authentication through multiple identity providers. Once authenticated, users are granted roles that define their access to AWS resources.

These services provide Booster with a scalable and secure backend platform for managing and storing data, devices, and authentication.

The next section focuses on the Rust programming language, and its use in AWS Lambda.

4 Rust Programming Language

The Rust programming language is a statically typed systems-level language, that emphasizes memory safety, performance and concurrency. It was created by Mozilla employee Graydon Hoare in 2006 as a response to memory safety issues in other systems-level languages like C and C++ [20]. Rust has since gained significant traction, winning the “most-admired progamming language” title in the Stack Overflow Developer Survey several years in a row, with the most recent win in 2024 [21]. Rust combines low-level performance control with high-level safety and convenience, without the need for a garbage collector or runtime [22]. Despite it being a systems-level language, Rust is used in different use cases such as CLI apps, web services, cloud computing, and performance-critical applications. This chapter will examine Rust’s key features, and why it is a strong candidate for Booster’s Lambda functions.

4.1 Rust features

Rust is similar to C and C++ in terms of performance and low-level control, but it introduces a safer approach to memory management. It overcomes common memory management issues by introducing features like ownership and borrowing. In addition, Rust forces immutability by default. This section will go through these key features.

4.1.1 Mutability

By default, variables in Rust are immutable. This means that the value of a variable cannot be changed once it has been defined. This feature promotes safety by preventing unintended changes to data. However, when a variable needs to be changed after its definition, it can be declared as mutable by using the mut keyword. An example of a mutable variable is shown in Figure 8.

```rust
fn main() {
  let mut x = 1;
  println!("x = {}", x);       // 'x = 1'

  x = 2;
  println!("x = {}", x);       // 'x = 2'
}
```

Figure 8. Example of a mutable variable in Rust.

While mutability makes it possible for variables to change their values, there are situations where a value should remain constant throughout the execution of the program. For these situations, Rust has constants. Constants are always immutable, and can never be changed after they are defined. They are declared with the const keyword and their type must always be explicitly specified, as depicted in Figure 9.

```rust
fn main() {
  const PI: f64 = 3.14;
}
```

Figure 9. Example of a constant declaration in Rust.

In addition to constants and mutability, Rust has a feature called shadowing. Shadowing is declaring a new variable with the same name as the previous variable. The new declaration replaces the previous one, allowing the reusage

of the variable name while potentially changing its type or value without mutating the original variable. This is demonstrated in Figure 10.

```
fn main() {
  let x = 1;
  println!("x = {}", x);        // 'x = 1'

  let x = x + 1;
  println!("x = {}", x);        // 'x = 2'

  let x = "hello";
  println!("x = {}", x);        // 'x = hello'
}
```

Figure 10. Example of shadowing in Rust.

4.1.2 Type system

In Rust, each value is of a specific data type, which must be known at compile time due to its static type system. Rust’s data types can be categorized into two subsets: scalar and compound.

Scalar types represent single values and include four primary types: integer, floating-point number, Boolean, and character.

An integer type represents a number without a fractional component. Rust supports both signed and unsigned integers. Signed integers can represent both positive and negative values, while unsigned integers represent only positive values. [23] Example shown in Table 1.

Table 1. Built-in Integer Types in Rust.

Unsigned			Signed		
Type	**Min. value**	**Max value**	**Type**	**Min. value**	**Max value**
u8	0	2^{8}-1	i8	-2^{7}	2^{7}-1
u16	0	2^{16}-1	i16	-2^{15}	2^{15}-1
u32	0	2^{32}-1	i32	-2^{31}	2^{31}-1
u64	0	2^{64}-1	i64	-2^{63}	2^{63}-1
u128	0	2^{128}-1	i128	-2^{127}	2^{127}-1

If an integer type is not annotated, Rust defaults to i32. In addition to the fixed-width integer types, Rust provides usize and isize, which are pointer-sized integers. This means that the exact size of these types depend on the target architecture: they are 32-bit on a 32-bit system, and 64-bit on a 64-bit system. An example of Rust's integer type annotation is shown in Figure 11.

```
fn main() {
  let x = 1;              // i32 (default)
  let y: i128 = 10000;    // i128 (annotated)
  let z: u8 = -1;         // error, unsigned values cannot be negated
  let q: i8 = 256;        // error, out of range for i8
}
```

Figure 11. Example of Rust's integer type annotation.

Floating-point number types represent signed numbers with decimal points. Rust provides two floating-point types, f32 and f64. The default type is f64 because it offers better precision, and performs roughly as fast as f32 on modern CPUs. [23] Rust's floating-point number type is demonstrated in Figure 12.

```
fn main() {
  let x = 1.0;              // f64 (default)
  let y: f32 = 1.0;         // f32 (annotated)
}
```

Figure 12. Example of Rust's floating type number annotation.

Boolean types in Rust are either true or false, just like in most other programming languages. The type is specified using the bool keyword, as seen in Figure 13.

```
fn main() {
  let x = true;             // bool (default)
  let y: bool = false;      // bool (annotated)
}
```

Figure 13. Example of Rust's Boolean type annotation.

The final type of the scalar subset is the character type. It is specified using the char keyword, and it is Rust’s most primitive alphabetic type. The character type can represent more than just ASCII, like Chinese or Russian letters and even emojis. The character type is depicted in Figure 14.

```
fn main() {
  let x = 'h';                  // char (default)
  let y: char = '诶';          // char (annotated)
  let z: char = '🦀';         // char (annotated)
}
```

Figure 14. Example of Rust's character type annotation.

Compound types are capable of grouping multiple values into a single type. Rust has two primitive compound types: tuples and arrays.

A tuple is a fixed-size collection of values of different types. Once a tuple is declared, its size cannot change. An example of this is in Figure 15.

```
fn main() {
  let x = (100, 2.0, true, '🦀');        // tuple (default)
  let y: (u8, f32, bool, char) = x;     // tuple (annotated)
}
```

Figure 15. Example of Rust's tuple type annotation.

The second and final primitive type of compound types is the array type. Arrays are another way of collecting values together. Arrays differ from tuples mainly that each element of an array must be of the same type. Like tuples, arrays in Rust have fixed length. The array type is depicted in Figure 16.

```
fn main() {
  let x = [1, 2, 3, 4, 5];      // array (default)
  let y: [i32; 5] = x;          // array (annotated)
  let z = [1; 3];               // array (annotated) = [1, 1, 1]
}
```

Figure 16. Example of Rust's array type annotation.

4.1.3 Ownership and borrowing

All programs need to manage the way they use a computer's memory during execution. Some languages rely on garbage collection, which automatically frees unused memory, others require manual allocation and freeing of memory. Rust takes a different approach: it manages memory through an ownership system with a set of rules that are checked at compile time. If any of the rules are broken, the program won't compile. [24]

The three rules of ownership, demonstrated in Figure 17:

- Each value in Rust has an owner.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped.

```
fn main() {
  // Rule #1
  let x = String::from("hello");  // x is now the owner of the String "hello"

  // Rule #2
  let y = x;  // y is now the owner of the String "hello"

  // println!("{}", x);  // compile time error, because ownership of the String "hello" has moved to y

  // Rule #3
  {
    let z = String::from("world");  // z is the owner of the String "world"

    // z goes out of scope here -> the String "world" is dropped
  }

  // y goes out of scope here -> the String "hello" is dropped
}
```

Figure 17. Example of Rust's ownership rules.

While ownership ensures that memory is managed safely, Rust also provides a mechanism for borrowing, which allows multiple parts of the code to access data without taking ownership of it. The actual mechanism that enables borrowing in Rust is the reference. References allow data to be borrowed either immutably or mutably.

Immutable references allow the borrower to only read the data, and prevent any modification. An example of immutable references is demonstrated in Figure 18.

```rust
fn main() {
  let x = String::from("hello");

  // borrow 'x' by passing a reference (&x) as the function argument
  let len = string_length(&x);

  println!("The length of {} is {}.", x, len);    // 'The length of hello is 5.'
}

// function that returns the length of a reference to a String
fn string_length(string: &String) -> usize {
  string.len()
}
```

Figure 18. Example of an immutable reference in Rust.

Mutable references allow both reading and modifying the data, but only one mutable reference can exist at a time, as shown in Figure 19.

```rust
fn main() {
  let mut x = String::from("hello");

  println!("Before: x = {}", x);          // 'Before: x = hello'

  // pass a mutable reference to the String as the function argument
  add_world(&mut x);

  println!("After: x = {}", x);           // 'After: x = hello world'
}

// function that modifies the String by adding ' world' suffix to it
fn add_world(s: &mut String) {
  s.push_str(" world");
}
```

Figure 19. Example of a mutable reference in Rust.

4.1.4 Cargo

Cargo is Rust’s build tool and package manager. It is used to automate dependency management, testing, document generation, and building. Cargo is

widely used for managing Rust projects, providing a standardized workflow and simplifying the development process.

Each Cargo project has a central configuration file, Cargo.toml, which contains metadata about the project, such as the package name, dependencies, version, and more. Cargo.toml also provides configurations for different build profiles, compilation features, and workspace settings.

Rust's ecosystem relies on reusable libraries called crates, which are hosted on the official Rust package registry, Crates.io. The usage of external crates are specified in the Cargo.toml file. When the Rust project is built, Cargo automatically downloads and compiles these dependencies, managing versions and compatibility.

4.2 Rust for AWS Lambda

AWS announced the general availability of the AWS SDK for Rust in late 2023. It offers an idiomatic, type-safe API that leverages the language’s key strengths. The SDK supports modern Rust features like async/await, non-blocking I/O, and builders. It includes a wide range of APIs, enabling interaction with over 300 AWS services, each packaged as its own crate. [25]

Example of a simple “Hello” Lambda function written in Rust is demonstrated in Figure 20.

```rust
// import libraries
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

// asynchronous handler function for processing Lambda events
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
  // extract the event payload from the function trigger
  let payload = event.payload;

  // get the first name from the message, if the field is empty, use 'world'
  let first_name = payload["firstName"].as_str().unwrap_or("world");

  // return a JSON response
  Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

// use the tokio runtime to enable asynchronous execution
#[tokio::main]
// the entry point function
async fn main() -> Result<(), Error> {
  // run the Lambda function using the handler and lambda_runtime library
  lambda_runtime::run(service_fn(handler)).await
}
```

Figure 20. "Hello" Lambda function in Rust.

This example Lambda function is designed to respond to an incoming trigger, which contains an optional field “firstName” in JSON format. If the field is not empty, the function extracts it and uses it in the response message. If the field is empty, the function defaults to greeting "world." The function then returns a JSON response containing the personalized greeting message.

Once a Lambda function is written, the next and final step is compiling and deploying it. There are several approaches to the deployment of Rust-based Lambdas, each with its own set of tools and workflows. Some common deployment methods include:

- Cargo Lambda: A Rust-specific tool designed to simplify deployment and packaging for AWS Lambda.
- AWS CLI: The AWS CLI that can be used for manually deploying Lambda functions.

- Serverless Framework: A popular framework that abstracts deployment tasks and helps manage serverless applications, including Lambda functions.

Since Booster's deployment architecture is Serverless-based, this thesis will use the Serverless framework as a deployment basis.

4.3 Challenges and trade-offs

When implementing existing Node.js based Lambda functions in Rust, there are some challenges and trade-offs that need to be considered.

One of the primary challenges is Rust's steep learning curve. Its approach to memory management can be difficult to grasp for developers used to garbage-collected languages like TypeScript. In addition, Rust's syntax is more complex compared to TypeScript, and figuring out which crates are needed for the new implementation can take some time.

This learning curve affects development speed, which is significantly slower compared to TypeScript, especially in the early stages of development. While development speed improves with experience, it is unlikely to match the speed of writing TypeScript. Rust's strict type system and rigorous compiler checks also slow down development, but they provide safer and more reliable code.

Another challenge is integrating Rust with a codebase that is written in TypeScript. Booster's architecture is built on TypeScript types and functions, can lead to compatibility issues when rewriting existing functions with Rust. In addition, some libraries used in the current solution might not have direct Rust equivalents.

5 Current Implementation

The three most performance-critical Lambda functions written in TypeScript are:

- devicesStatesGet
- organizationsGet
- usersGet

All of these Lambda functions operate behind GraphQL APIs to query data.

Each of these Lambda functions has a wrapper function on top of the actual handler function. This wrapper (createBoosterGqlHandler) is responsible for handling the GraphQL requests, logging, and error management. It acts as a middleware between the GraphQL API and the core logic in Lambda functions.

5.1 devicesStatesGet

The devicesStatesGet Lambda function is responsible for fetching and returning the state of a certain device. It starts by checking if the user has access to the device. If the user has invalid access, it throws an error. If the user has valid access, the function next fetches the device's shadow, connection state, latest timestamp, and metadata. Finally, the Lambda function constructs the device state from the data and returns it. If the device has no shadow, the function returns null. The functionality and flow of the Lambda is illustrated in Figure 21.

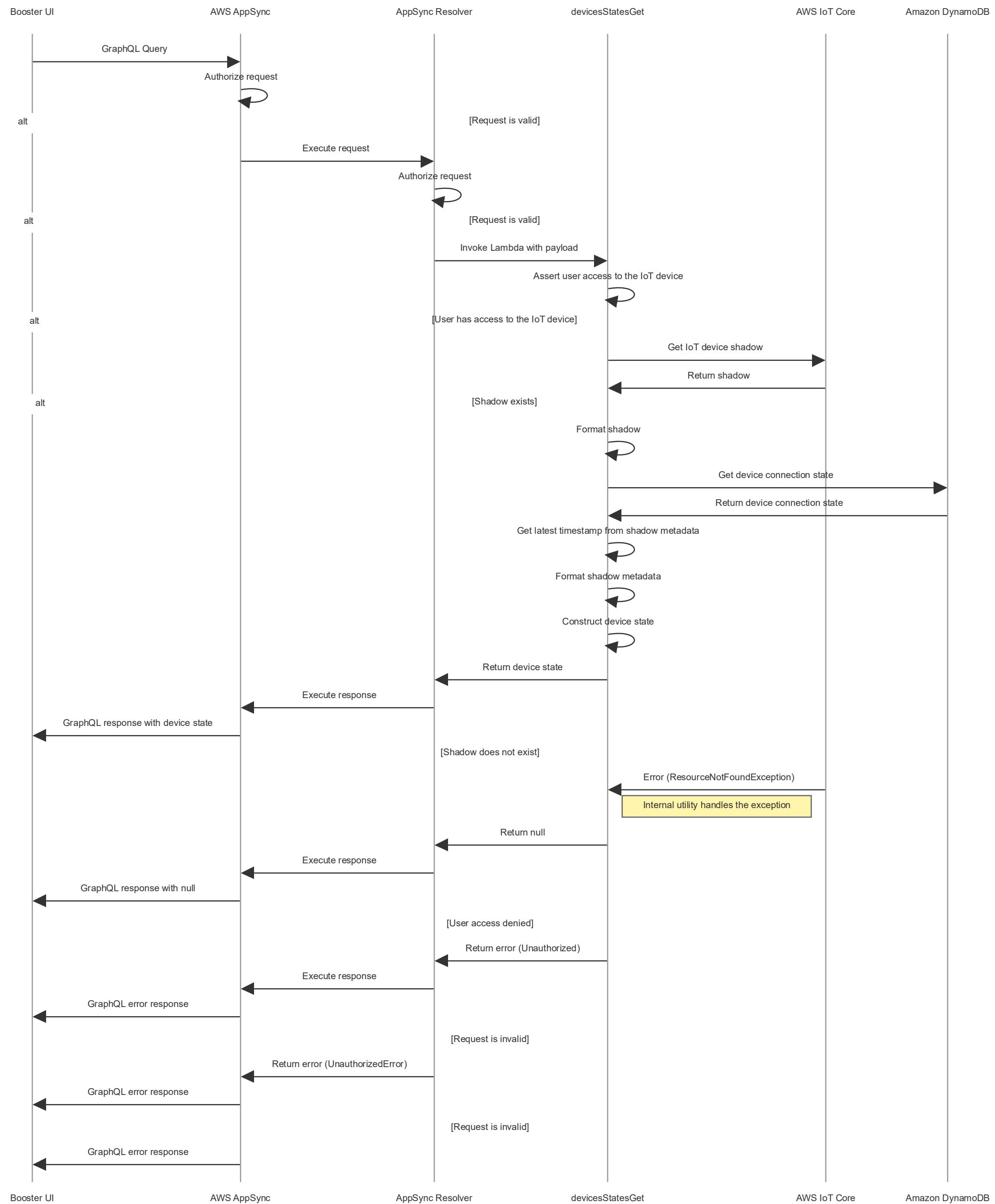

Figure 21. Sequence diagram of the devicesStatesGet Lambda function.

5.2 usersGet

The usersGet Lambda function is responsible for fetching and returning the details of a specific user. It gets invoked by a payload containing the ID of the target user, first validating the ID. If the ID is invalid, return an error. If the ID is valid, the Lambda fetches the user entity from the entity database and user

details from Amazon Cognito’s user pool simultaneously. Once the user data is fetched, the Lambda checks that data exists. If yes, the Lambda asserts user and organization access, constructs an user object with the user data and returns it. If any of the data fields in the user data are missing, the internal utility functions the Lambda uses will throw an error. The Lambda’s sequence diagram is depicted in Figure 22.

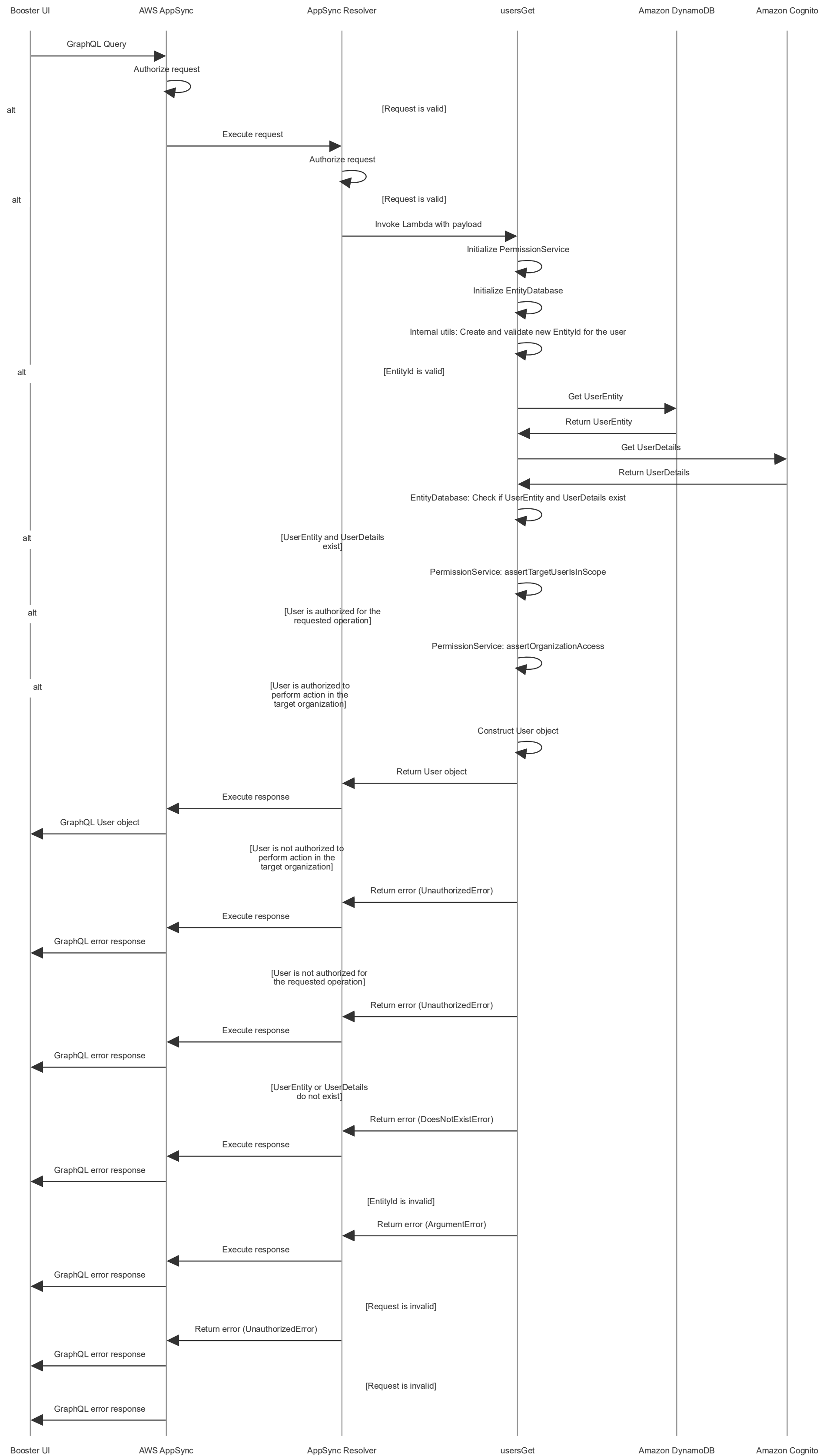

Figure 22. Sequence diagram of the usersGet Lambda function.

5.3 organizationsGet

The organizationsGet Lambda function works similar to usersGet. It gets invoked by a payload containing an organization ID, which it first validates. If the ID is not valid, the Lambda returns an error. If the ID is valid, the function asserts organization access, if invalid, an error is returned. If the user has valid organization access, it retrieves the organization entity. If the entity exists, it finally returns the organization object. If the organization does not exist, an error is returned. The Lambda's sequence diagram is shown in Figure 23.

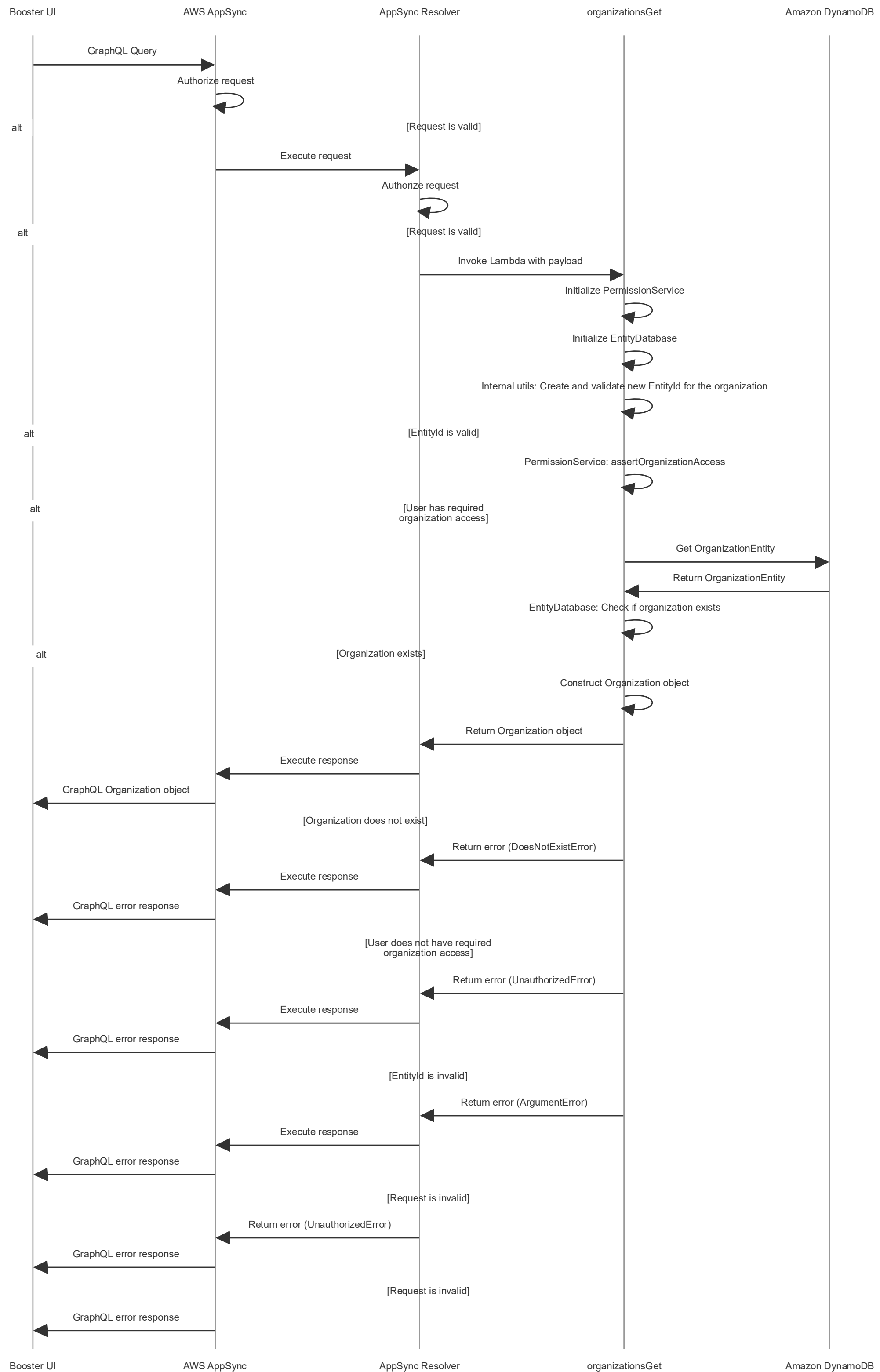

Figure 23. Sequence diagram of the organizationsGet Lambda function.

6 New Implementation

The process of implementing the Lambda functions in Rust involved translating existing functionality from TypeScript while adapting Rust’s stricter type system and tooling. This required adjustments to how types, utilities, and handlers were written. This chapter describes how the functions were implemented in Rust and how they were integrated into the existing codebase, as well as the development efficiency of the entire process.

6.1 Codebase structure

The Rust implementations followed the same structural conventions as their TypeScript counterparts. Each Lambda function was placed in its corresponding microservice’s “functions” directory, and implemented as an independent Cargo package named after the function. For example, devices_states_get Lambda was located at ./services/devices/functions/devices_states_get/, with a main.rs file for the handler, and a types.rs file for its associated types. This convention was followed for all three Lambdas.

Common utility logic and type definitions were implemented in a shared Rust module, located at ./common/src/rust_utils/. Only the relevant parts of the TypeScript utils and types directories were recreated.

In addition to the shared rust_utils module, the users service required some of its own internal utilities. These were implemented as a separate module at ./services/users/functions/service/rust_utils_users/, following the TypeScript structure. Although the structure of the files differ, the internal logic remains the same across both implementations.

Each Lambda was also configured in the respective functions.yml file, defining its handler name, runtime environment, and build artifact path. This setup made it possible to deploy the Rust-based Lambdas using Booster's existing Serverless infrastructure.

6.2 Implementation of Rust-based Lambda functions

Since the context of this thesis is focused on testing the new Rust-based Lambda functions, the createBoosterGqlHandler wrapper function was not implemented. The payloads used in the tests are always valid, so full request validation and error handling were not necessary.

In the original Node.js implementation, many types were generated directly from the GraphQL schema and used throughout the codebase. In Rust, these types had to be rewritten as struct definitions by hand.

The Lambda functions were developed and built with Cargo Lambda, a CLI tool for building and deploying AWS Lambda functions in Rust. Each Lambda was first tested locally using Cargo Lamba's emulator, and sending a test payload using the invoke subcommand. This test payload was created inside each Lambda package.

Finally, the Lambdas were compiled with Cargo Lambda. The builds were done in release mode to enable compiler optimizations and reduce binary size. The outputs were packaged as ZIP files, and the architecture was ARM64 to match the configuration of the Node.js-based Lambdas.

As mentioned earlier, each Lambda was configured so it could be deployed to the cloud using Serverless. However, the compilation step is not automated, so any changes to the Rust code require manually rebuilding the Lambda packages before deploying.

When compiling with Cargo Lambda, it automatically names the compiled Rust package's ZIP file as "bootstrap". If there are ZIP files with the same name, even if they are in different directories, the Lambdas will have the same SHA256 hashes, meaning that they are the same Lambda function. In the context of this thesis, the functions were renamed manually before deployment.

6.3 Development efficiency

One of the main points of this thesis was to evaluate the development efficiency of rewriting existing AWS Lambda functions with Rust. This section goes through the challenges and benefits experienced during the rewriting process.

As previously mentioned, Rust is known for its steep learning curve. Implementing the new Rust-based Lambda functions required learning and applying Rust's core concepts, which are not present in languages like TypeScript. This initial learning phase slowed the development progress, especially in the early stages. However, Rust's comprehensive documentation supported the process.

The Rust compiler enforces strict rules. Although this increased the time debugging and resolving compilation errors, it also reduced the occurrence of runtime bugs. In most cases, once a code compiled successfully, it also executed successfully.

The build process, especially in release mode with optimizations enabled, introduced further delays. This was particularly relevant during the testing phase, where frequent builds were required. The Lambdas' ZIP file sizes were also significantly larger compared to Node.js-based Lambdas.

Although the core logic of the Lambda functions remained the same, translating them from TypeScript to Rust was not a direct process. The differences in syntax, language constructs, types, generated types and available libraries meant that the original solutions had to occasionally be done differently. Additionally, the TypeScript-based codebase included some complex design patterns. Reimplementing these patterns in Rust introduced challenges and required additional consideration.

Despite these challenges, the overall development experience with Rust significantly improved over time. Writing, testing, and deploying the first Lambda function took several weeks, primarily due to the learning curve and the need to build the foundational modules from scratch. In contrast, the final Lambda

function was completed in just one day. This was due to the increased familiarity with the language and tooling, but also because the layout and most of the needed utilities were already developed.

Going forward, if more Booster Lambdas are rewritten in Rust, the development process can be expected to become faster and smoother after the initial learning curve. AI tools could further help accelerate the implementation process and reduce the development effort. In the next section, the performance and costs of the Rust-based Lambdas are evaluated and compared to their TypeScript counterparts.

7 Testing

The testing of the Node.js and Rust-based Lambda functions were done with a script written in JavaScript, that invokes the Lambdas with a specific payload, with the amount of invokes and the interval between the invokes being configurable. Another script was also created with JavaScript, which initializes one mock device to AWS IoT Core with an X.509 certificate, so it could communicate with the Booster cloud so that there is data to handle for devicesStatesGet Lambda. A user and an organization were also created for the tests.

Amazon Cloudwatch logs were used to track the details of the Lambda runs, which were then logged in an Excel table for further analysis. Each Lambda's cold starts and warm starts were measured separately. The test set included 15 cold start invocations per Lambda with 10 minutes between each invocation. The warm starts had 30 invocations with a 2 second interval. The tables had calculated fields for the averages of duration, billed duration, maximum memory used, and initialization duration of each run.

Each Rust-based Lambda function was tested with different memory amounts to find out what amount would be optimal for performance and cost. The tested memory amounts were 128 MB, 256 MB, 384 MB, 512 MB and 1024 MB.

7.1 Performance

The performance section analyzes the execution durations and memory usage for both the Node.js and Rust-based Lambda functions. The Rust functions were tested with varying memory amounts, and the Node.js functions were tested only with the default memory amount of 1024 MB.

The average of performance results of all three Lambda functions are shown in Table 2, with the memory amount set for 1024 MB by default for both Node.js- and Rust-based Lambdas.

Table 2. Lambda performance comparison with 1024 MB memory amount.

Lambda	Event	Implementation	Duration (ms)	Maximum memory used (MB)	Initialization duration (ms)	Total duration (ms)
devicesStatesGet	Cold start	Node.js	289.56	116.13	535.61	855.17
		Rust	172.35	30.4	43.11	215.46
	Warm start	Node.js	63.85	125.43	-	-
		Rust	26.35	31.67	-	-
usersGet	Cold start	Node.js	336.6	115.67	530.85	872.85
		Rust	296.3	30.8	45.06	341.36
	Warm start	Node.js	140.2	117.67	-	-
		Rust	132.6	33	-	-
organizationsGet	Cold start	Node.js	139.68	115	514.59	654.27
		Rust	94.01	28.8	41.13	135.14
	Warm start	Node.js	10.99	117	-	-
		Rust	3.75	30.3	-	-

As seen in Table 2, the Rust-based Lambdas constantly outperformed their Node.js counterparts, particularly in terms of cold start initialization and memory usage. Cold start initialization durations were over 90% faster, and peak memory usage was reduced by over 70% in all three Lambdas. Execution times were noticeably faster in devicesStatesGet and organizationsGet, 30–50% faster, while the improvement in usersGet was smaller, around 13%.

The smaller performance gain in usersGet is due to a delay related to the API used to fetch the user details from the Cognito user pool. This call made up around 70% of the function’s execution time. This trend persisted across all memory configurations for usersGet.

Although usersGet had a smaller improvement in the execution duration, when examining the Lambda’s total duration, the Rust version was about 61% faster. This is because of Rust’s fast cold start initializations. The total duration was around 74% faster in devicesStatesGet and nearly 80% faster in organizationsGet.

The Rust-based Lambdas outperformed the Node.js ones under warm start conditions aswell. The devicesStatesGet function was nearly 59% faster, while usersGet was only marginally faster. The Node.js-based organizationsGet was already very fast, but the Rust version performed nearly 66% faster.

In addition to the fixed memory amount of 1024 MB, the Rust-based Lambdas were also tested with different memory amounts to evaluate what would be the optimal memory amount for each Lambda function relative to performance. The results are shown in Table 3.

Table 3. Rust-based Lambda performance comparison with different memory amounts.

Lambda	**Memory (MB)**	**Cold start duration (ms)**	**Cold start maximum memory used (MB)**	**Cold start initialization duration (ms)**	**Warm start duration (ms)**	**Warm start maximum memory used (MB)**
devicesStatesGet	128	876.42	28	47.66	30.70	29
	256	446.90	28	45.94	29.21	28.90
	384	310.07	29.40	44.66	28.35	31.40
	512	254.25	29.80	44.04	32.58	31
usersGet	128	915.55	28	46.76	139.71	28.50
	256	525.36	27.20	47.06	133.03	28.50
	384	479.17	29.40	44.40	140.26	31.60
	512	369.41	30.20	43.85	144.03	31
organizationsGet	128	693.65	26.60	42.98	6.03	27.10
	256	337.51	26.80	44.38	3.08	27.30
	384	217.72	28.80	39.81	3.36	29.80
	512	167.85	28.60	40.78	3.52	30.60

As seen in Table 3, lower memory amounts result in significantly longer cold start durations. In contrast, the warm start durations remain relatively stable across the different memory sizes, showing that smaller memory amounts do not negatively affect performance during warm starts. Additionally, the initialization duration and the maximum memory usage remain consistent with the different memory amounts.

The next section examines the cost implications of these memory configurations in more detail and compares the runtime costs of Node.js and Rust-based Lambdas.

7.2 Costs

The cost of a Lambda function is determined by several factors: the region in which it is deployed, the underlying CPU architecture, the number of invocations, the amount of memory allocated, and the function's billed duration. All Lambda functions in this thesis are deployed in the eu-central-1 region and use the ARM architecture.

While AWS offers a free tier that includes one million requests and 400,000 GB-seconds of compute time per month, the calculations in this thesis were done excluding the free tier. With the ARM architecture, the cost of running a Lambda is $0.0000133334 per GB-second. The cost is calculated as follows:

$$Cost = \left(\frac{Number\ of\ requests}{1{,}000{,}000} \times \$0.20\right) + (Number\ of\ requests \times Duration\ (ms) \times Memory\ (GB) \times \$0.0000000133334\ per\ GB - ms)$$

The memory is converted to GB using the exact formula:

$$1\ MB = 0.0009765625\ GB$$

The following calculations include the Lambdas with 1024 MB of allocated memory, and assume 1 million requests over a month. The formula simplified:

$$Cost = \$0.20 + (1{,}000{,}000 \times Duration\ (ms) \times 1\ GB \times 0.0000000133334)$$

Using this formula, Table 4 shows the billed durations and estimated costs for the Node.js and Rust-based Lambda functions during both cold and warm starts.

Table 4. Lambdas’ billed durations and estimated costs comparison with 1024 MB memory amount.

Lambda	Event	Implementation	Billed duration (ms)	Cost ($)
devicesStatesGet	Cold start	Node.js	290	4.07
		Rust	216	3.08
	Warm start	Node.js	65	1.07
		Rust	27	0.56
usersGet	Cold start	Node.js	367	5.09
		Rust	342	4.76
	Warm start	Node.js	141	2.08
		Rust	133	1.97
organizationsGet	Cold start	Node.js	141	2.08
		Rust	136	2.01
	Warm start	Node.js	12	0.36
		Rust	4	0.25

Even though the Rust-based Lambdas were faster in execution time, the difference of billed durations compared to the Node.js-based Lambdas is smaller than expected. This is due to the way AWS bills custom runtimes like Amazon Linux 2023, which was used in the Rust functions. For custom runtimes, the billed duration includes both the initialization duration and the function’s execution duration during cold starts. Managed runtimes like Node.js do not include the initialization phase in the billed duration. During warm starts, only the execution duration is billed for both runtimes.

The devicesStatesGet function shows the most significant cost savings with Rust. It is about 24% cheaper on cold starts and nearly 48% cheaper on warm starts compared to Node.js. The rest of the functions have much smaller cost differences between the implementations.

The same approach was used to estimate the costs over a month for the different memory amounts by adjusting the memory value accordingly. The results are shown in Table 5.

Table 5. Rust-based Lambdas' billed durations and estimated costs comparison with different memory amounts.

Lambda	Memory (MB)	Event	Billed duration (ms)	Cost ($)
devicesStatesGet	128	Cold start	925	1.74
		Warm start	32	0.25
	256	Cold start	494	1.85
		Warm start	30	0.30
	384	Cold start	356	1.98
		Warm start	29	0.34
	512	Cold start	299	2.19
		Warm start	34	0.43
usersGet	128	Cold start	963	1.80
		Warm start	141	0.43
	256	Cold start	573	2.11
		Warm start	134	0.65
	384	Cold start	525	2.83
		Warm start	141	0.90
	512	Cold start	414	2.96
		Warm start	145	1.17
organizationsGet	128	Cold start	738	1.43
		Warm start	7	0.21
	256	Cold start	383	1.48
		Warm start	4	0.21
	384	Cold start	259	1.49
		Warm start	4	0.22
	512	Cold start	209	1.59
		Warm start	5	0.23

The cost difference is directly in relation with the memory amount. Even though the billed durations were higher with smaller memory amounts, the overall costs were still lower. Given these trade-offs, the next section focuses on choosing the best compromise between performance and cost for the Rust-based Lambdas.

7.3 Optimal memory configuration

As seen from the test results, the amount of memory allocated to a Lambda function affects both its performance and costs. Therefore, it is important to determine the optimal memory configuration for each Rust-based Lambda function.

The optimal memory configurations were chosen by comparing each function's performance and cost with the original 1024 MB Node.js implementation. The goal was to optimize performance while minimizing costs. The results are shown in Table 6.

Table 6. Optimal Rust-based Lambdas' memory configurations between performance and cost.

Lambda	**Memory amount (MB)**	**Cold start duration (ms)**	**Cold start initialization duration (ms)**	**Cold start cost ($)**	**Warm start duration (ms)**	**Warm start cost ($)**
devicesStatesGet	512	254.25	44.04	2.19	32.58	0.43
usersGet	512	369.41	43.85	2.96	144.03	1.17
organizationsGet	512	167.85	40.78	1.59	3.52	0.23

Allocating 512 MB of memory to each Rust-based Lambda function found to be the most optimal memory configuration between performance and costs. The

Rust functions achieve cold start total durations significantly lower than their Node.js counterparts.

The total cold start duration of devicesStatesGet drops from approximately 825 ms in Node.js to around 298 ms, a reduction of nearly 64%. The monthly costs for a million requests also drop about 46%, from $4.07 to $2.19.

Even though usersGet was slower compared to the other Rust-based Lambdas, with this memory amount, its total cold start duration is around 52% faster, with Node.js being at 867.45 ms and the Rust version having a total execution duration of 413.26 ms. The costs come down from $5.09 to $2.96, a near 42% drop.

The organizationsGet Lambda has a total execution duration of 208.63 ms, compared to the 654.27 ms of the Node.js version, a 68% drop. The costs also come down from $2.08 to $1.59, a smaller drop of 23%.

As examined before, the memory amount mostly affects the cost, the cold start initialization duration, and naturally the cold start total duration. Warm start durations and maximum memory usage stay consistent regardless of the Lambda's allocated memory size.

8 Conclusion

The purpose of this thesis was to rewrite the most performance-critical Node.js-based Lambda functions in Rust, and compare their performance, cost and development efficiency. Three Lambda functions were reimplemented in Rust; devicesStatesGet, usersGet, and organizationsGet. The main target was to understand if a larger rewrite of the Booster IoT Asset would make sense, where at least the performance-critical parts would be reimplemented in Rust.

The performance differences were noticeable between the Node.js-, and Rust-based functions with the default memory amount of 1024 MB, especially with the cold start initialization duration and memory usage. Rust showed a consistent 90% drop in initialization duration, and 70% smaller amount of maximum memory used in each Lambda function. This led to lesser or similar costs, partly due to the way AWS Lambda bills custom runtimes. The Rust functions were also tested with different memory amounts, and 512 MB of allocated memory was found to be the most optimal memory amount for each Lambda between performance and costs.

Based on the results, rewriting performance-critical Node.js-based Lambda functions in Rust in Booster can be beneficial in both performance and costs, despite Rust's initial learning curve. The speed of learning Rust's different concepts depends on the person, but the rewriting phase can be done faster with the use of AI tools.

The potential benefit of rewriting depends on the context of the Lambda function; if the function's execution time is mostly spent on waiting on API calls, the performance improvements will most likely be small. In these cases, the execution duration might not benefit from Rust, the cold start initialization duration will.

For the Rust-based versions to be implemented into Booster, some additional work is needed. Since this thesis focused only on the performance, costs and development efficiency, there were no automation pipelines or tests

implemented for the Rust-based Lambdas. The functions were not integrated with AppSync either, since the testing was done directly in the AWS Lambda environment. In addition, the functions' createBoosterGqlHandler wrapper was not implemented in Rust. The current Node.js implementation uses code generation to translate GraphQL schemas into types – since there is no suitable Rust crate for this, the type conversions had to be done manually.

References

[1] Susnjara S., Smalley I. “Serverless”, IBM, 2024. [Online]. Available: https://www.ibm.com/think/topics/serverless. Accessed: 18.02.2025

[2] Mattfield L. “Comparing Lambda Runtime Performance”, Commerce Architects, 2024. [Online]. Available: https://www.commerce-architects.com/post/comparing-lambda-runtime-performance. Accessed: 18.02.2025

[3] Crosland C. “Serverless Speed: Rust vs. Go, Java, and Python in AWS Lambda Functions”, Scanner.dev, 2023. [Online]. Available: https://blog.scanner.dev/serverless-speed-rust-vs-go-java-python-in-aws-lambda-functions/. Accessed: 18.02.2025

[4] Smith T. “AWS Lambda Benchmarking”, Xebia, 2024. [Online]. Available: https://xebia.com/blog/aws-lambda-benchmarking/. Accessed: 18.02.2025

[5] SADE Innovations. “SADE Booster IoT Asset”, 2025. [Online]. Available: https://sadeinnovations.com/iot. Accessed: 12.03.2025

[6] SADE Innovations. ”Booster IoT Asset Overview”, 2024. [Online]. Available: https://sadeinnovations.com/doc/en/SADE%20Booster%20overview_public.pdf. Accessed: 12.03.2025

[7] Amazon Web Services. “Understanding the Lambda execution environment lifecycle”, 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html. Accessed: 18.03.2025

[8] Amazon Web Services. “What is AWS AppSync?”, 2025. [Online]. Available: https://docs.aws.amazon.com/appsync/latest/devguide/what-is-appsync.html. Accessed: 18.03.2025

[9] Amazon Web Services. “GraphQL and AWS AppSync architecture”, 2025. [Online]. Available: https://docs.aws.amazon.com/appsync/latest/devguide/graphql-overview.html. Accessed: 18.03.2025

[10] Amazon Web Services. “GraphQL Schemas”, 2025. [Online]. Available: https://docs.aws.amazon.com/appsync/latest/devguide/schema-components.html. Accessed: 18.03.2025

[11] Amazon Web Services. “Data sources”, 2025. [Online]. Available: https://docs.aws.amazon.com/appsync/latest/devguide/data-source-components.html. Accesesed: 18.03.2025

[12] Amazon Web Services. “Resolvers”, 2025. [Online]. Available: https://docs.aws.amazon.com/appsync/latest/devguide/resolver-components.html. Accessed: 18.03.2025

[13] Amazon Web Services. “Improving data access with secondary indexes in DynamoDB”, 2025. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html. Accessed: 20.03.2025

[14] Amazon Web Services. ”Partitions and data distribution in DynamoDB”, 2025. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html. Accessed: 20.03.2025

[15] Amazon Web Services. “AWS IoT Core FAQs”, 2025. [Online]. Available: https://aws.amazon.com/iot-core/faqs/. Accessed: 24.03.2025

[16] Amazon Web Services. “Device communication protocols”, 2025. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html. Accessed: 24.03.2025

[17] Amazon Web Services. “AWS IoT Device Shadow service”, 2025. [Online]. Available: http://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html. Accessed: 24.03.2025

[18] Amazon Web Services. “AWS IoT security”, 2025. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/iot-security.html. Accessed: 24.03.2025

[19] Amazon Web Services. “What is Amazon Cognito?”, 2025. [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html. Accessed: 30.05.2025

[20] Thompson C. “How Rust went from a side project to the world’s most-loved programming language”, MIT Technology Review, 2023. [Online]. Available: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/. Accessed: 08.04.2025

[21] Stack Overflow. “2024 Developer Survey”, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology#admired-and-desired. Accessed: 08.04.2025

[22] Rust Blog. “Announcing Rust 1.0”, 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html. Accessed: 08.04.2025

[23] The Rust Programming Language Book. “Data Types”, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch03-02-data-types.html. Accessed: 08.04.2025

[24] The Rust Programming Language Book. “What Is Ownership?”, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accessed: 08.04.2025

[25] Amazon Web Services. “Announcing general availability of the AWS SDK for Rust”, 2023. [Online]. Available: https://aws.amazon.com/blogs/developer/announcing-general-availability-of-the-aws-sdk-for-rust/. Accessed: 08.04.2025