CHALMERS
UNIVERSITY OF TECHNOLOGY

NewWave
GROUP

Automation of Azure Test Environment with Microsoft Teams

Cost-Effective Instances Management using Webhooks and Azure Pipelines

Degree project report in Computer Science

Attila Lundin
Shahzaib Syed Kazmi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Degree project report 2024

Automation of Azure Test Environment with Microsoft Teams

Cost-Effective Instances Managment using Webhooks and Azure Functions

Attila Lundin
Shahzaib Syed Kazmi

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

Automation of Azure Test Environment with Microsoft Teams
Cost-Effective Instances Managment using Webhooks and Azure functions

Supervisor: András Kovács, CSE
Examiner: Jonas Duregård, CSE

Degree project report 2024
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: New Wave Group's logo.

Typeset in LaTeX
Gothenburg, Sweden 2024

Automation of Azure Test Environment with Microsoft Teams
Cost-Effective Instances Managment using Webhooks and Azure functions
Attila Lundin © Shahzaib Syed Kazmi, 2024.
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The project developed a user-friendly application for managing Azure test environment instances for New Wave Group. The aim was to reduce cost associated with the utilisation of each instance and increase the efficiency with regards to development processes. Currently, at New Wave Group, these instances are constantly operational, even though they are only needed for approximately two days a month. This continual operation results unnecessary costs for the company.

By integrating an outgoing webhook with a Microsoft Teams channel to start, delete, and schedule instances, we ensured a user-friendly and cost-effective solution, with expenses directly tied to usage time. The effectiveness was evaluated through extensive testing using two main methods, user and integration tests. The tests were conducted using ngrok to facilitate application testing that requires external hosting. The final outcome of the project indicates significant monetary savings if the application were to be integrated into the existing infrastructure. It also reduces the complexity and the number of steps required to manage the resources.

Keywords: Azure functions, RESTful API, Microsoft Teams, CLI, Automation, Resources, Pipelines.

Acknowledgements

We would like to extend our deepest gratitude to Rozgar Khatab and Peach Larsson, who were not only great colleagues but also classmates, for their hard work on a related part of the project. Their efforts were crucial for our success. We would also like to thank András Kovács who was our supervisor at Chalmers University of Technology. His advice was essential for the project. A special thanks goes to Argon Omarson, Jimmy Balderud and Erik Olausson from New Wave Group for their assistance througout the project.

Attila Lundin and Shahzaib Kazmi, Gothenburg, June 2024

List of Acronyms

API	Application Programming Interface
CI	Continuous integration
CLI	Command Line Interface
GB	Giga Byte
GDPR	General Data Protection Regulation
GUID	Global Unique Identifier
HTTP	Hyper text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
JSON	JavaScript Object Notation
LRS	Local Redundant Storage
SEK	Swedish Krona
TLS	Transport Layer Security
URL	Uniform Resource Locator
URI	Uniform Resource Identifiers
YAML	Yet Another Mark up Language

Contents

List of Figures

Listings

1
Introduction

In this work we report on our progress in improving automation for the development processes in New Wave Group AB, building on the current solutions in Microsoft Azure. The main goal is to develop user-friendly infrastructure that enables starting and shutting down test environments by typing commands in a Microsoft Teams channel.

1.1 Background

New Wave Group AB is a company whose operations revolve around the acquisition and development of companies, their products, and brands [3] with a portfolio that consists of 43 brands such as Tenson, Orrefors, and Kosta Boda [4]. New Wave Group has an integrated supply chain [5] and a shared e-commerce platform with separate interfaces for each brand. The IT infrastructure therefore needs to support functions such as inventory management, financial transactions, orders, and logistical flows. The business system was previously hosted on servers which were on premise but has recently migrated its digital operations to the cloud service Microsoft Azure.

The development process used by the company includes unit and integration testing for its codebase. Testing is currently done in a mirror image of the production environment, a configuration which results in a number resources being rented on a monthly bases. This solution costs the company approximately SEK 34,000 per month. Currently, the development environment is used for just over two days a month for testing purposes, which according to Jimmy Balderud, Tech Lead at New Wave Group, is a very low utilisation rate. Despite the limited use the company currently pays for the environment as though it was in constant use. However, the reality is that it is used for less than 10 percent of the month and leads to unnecessary costs. To address this, the company wants to shut down resources when they are not in use, reducing waste and saving money. The company therefore wants to take a step towards continuous integration (CI) [6] and automate the process of resource start and shutdown.

New Wave Group is interested in using a restricted Microsoft Teams channel as a command line interface to control resources in the development environment. The main reason is ease of use since Teams is the primary communication channel for the development staff and it would reduce the number of interfaces developers need interact with. The idea is that a single command will suffice to manage a resource without need to interact with Azure.

1.2 Purpose

The primary purpose of this project is to minimise New Wave Group's cloud expenditure by designing, evaluating, and implementing a user-friendly and safe solution for managing the company's Azure test environment instances.

1.3 Objectives

The main goal of the project aims to improve cloud resource management at New Wave Group. The final solution should be secure, easy to use and maintain. To achieve this the following technical objectives should be met:

1. Implementation:
   (a) Develop a program that enables automatic management of Azure test environment instances.
   (b) The program should receive commands from Microsoft Teams via a webhook to facilitate real-time communication.
   (c) The program should be designed to execute an Azure CLI operation in response to specific events.
2. Evaluation
   (a) Ensure the reliability and security of the program in various environments and situations.
   (b) Ensure the efficiency of the solution.
3. Documentation.
   (a) All code should be thoroughly commented.
   (b) This report acts as comprehensive documentation for the project.

1.4 Limitations

The project is limited to only manage the automation, including scheduling, of start and delete, for three Azure resource groups : genericCms, genericAPI and genericShop. Additionally, unit tests were excluded.

2
Technical Background

This chapter explains technologies and concepts that are integral to the project, offering a extensive overview and thorough analysis of each. The introductory section will present continuous integration (CI) followed by an summary of the Hyper Text Transfer Protocol (HTTP) and an explanation of webhooks. Thereafter a select number of tools from the Microsoft development ecosystem and suite will be explained.

2.1 Continuous Integration (CI)

Continuous integration (CI) is a methodology which attempts to streamline the development process [7]. The method is not a defined technological stack, rather an interplay of technologies and working practices [8]. CI is the notion of incremental additions to a codebase, which is made possible by automation. CI increases the frequency of evaluation which helps to assure compliance [9], quality and reliability of a codebase [7, 6, 10] all while being a time-saving measure [8].

2.2 Hyper Text Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) [11] and Hypertext Transfer Protocol Secure, also known as HTTP over TLS, (HTTPS) [12] are protocols used for communication between web browsers and web servers [13]. What differentiates HTTPS from HTTP is that data is sent in an encrypted manner while over HTTP, where simplicity is preferred, data is sent as plaintext [12]. An HTTP packet consists of two sections, the header and the body [13]. The header contains all the necessary information for the packet to be processed correctly, e.g. protocol version [14]. The headers differ based on whether it is a part of a request or a response, i.e. an acknowledgement to a request [14]. The body on the other hand is a section in the packet which carries a payload [13, 15]. Not all requests contain a body, but those that do often format the data in a JavaScript Object Notation (JSON) [15].

The protocol follows client- server architecture where communication between each actor is accomplished by sending requests [13]. There are a total of five request methods in which all serve distinct purposes [16]. These can range from retrieving, uploading and deleting data. The initial request method is referred to as a GET whereas the others are referred to as POST and DELETE respectively. The GET request method is used for retrieving data [17]. If the request is successful in retrieving the desired payload, a status code of 200 is returned [17, 15]. The GET request does not have a body [17]. On the other hand, the POST method is used to upload data to the server and it is done by assembling a request body with the data a client, or a server wants to transmit [18]. Lastly the DELETE method is used for deletion of a certain resource in the server and like the GET method does not contain a request body with a payload [19].

2.3 Webhooks

A webhook facilitates communication between applications in real time through the use of triggers and events [20, 21]. It works by having a specific event occurring in one application triggering the automatic data transmission to the other, in a one-way flow [20, 21]. Essentially, this means that as soon as a specific event occurs, it allows the program to send data in a unidirectional fashion. Data that is sent from the source to the target application have no immediate responses. To support communication in both directions, there are two types of webhooks; incoming and outgoing [20, 21, 22, 23]. An incoming webhook, as the name suggests, is a mechanism designed to allow one application to receive real time data sent from another external application after a specific event. This is achieved by listening to incoming HTTP traffic. An outgoing webhook is just like an incoming webhook, an HTTP POST request is generated by a program during an event, which is then sent to an endpoint [20, 21, 22].

2.4 Ngrok

Ngrok is a tool that allows developers to test applications by creating a secure tunnel to expose their local servers to the internet. In essence, this tool forwards HTTP packets to a specified port on the local machine. This tool is particularly useful for testing applications because there is no need to deploy to a production environment. Ngrok simulates the application's behaviour in a real production environment by rerouting packets to localhost [24, 25, 26].

2.5 Microsoft suite and Development Ecosystem

Microsoft provides an extensive array of services, encompassing both proprietary [27] and open-source, freely available technologies [28]. The following subsections will explore the specific Microsoft technologies that were employed in the project.

2.5.1 C#

C#, pronounced as C-sharp, is a programming language developed by Microsoft as a part of its .NET framework [29]. The language has its roots in the C family but will still get recognized by developers who are familiar with either Java or JavaScript [29]. C#, however, is a modern, object-oriented, and type safe language used for building various types of applications [29]. Its capabilities extend across the creation of web applications, mobile apps and cloud services [29].

2.5.2 Microsoft Teams

Microsoft Teams is a proprietary communications platform which is mainly targeted towards professional use [30]. Microsoft offers deep integration with the office suite and other services such as Azure DevOps [31]. Each user is assigned to an organisation, which can be seen as a collective workspace for the workforce. The users are additionally a part of one or more teams. Said users can interact with other members of a team in a so-called channel. Messages sent in a channel are visible to all users that are a member of the team [32].

Outgoing webhooks are natively supported in and is created as any other Teams App for a certain Teams team [33]. During the initialization, each outgoing webhook is assigned a name and a destination address. When the webhook is created it will appear as a user within the Teams team [22, 34].

The triggering of the outgoing webhook is done by user interactions. Teams will parse the input from the user and send it to the destination address when a reference to the webhook is made in the message. All the formatting of the user input will be kept in the request [34].

2.5.3 Azure DevOps

Azure DevOps is a platform created by Microsoft, as a Software as service (SaaS) [35] solution, in which various tools are rendered for the efficient development and deployment of software projects [36]. It also includes a set of features such as project management tools (Azure boards), version control (Azure repos), continuous integration and continuous delivery (Azure pipeline) and many more offerings [37]. The main objective of Azure DevOps is to optimise the software development process, enhance the collaboration between the development and operational teams [36]. It additionally help teams to streamline the entire lifecycle of software development [36].

2.5.4 Tasks and Pipelines in Azure

Azure pipelines are cloud-based services designed to automate the process of building and testing projects which thus facilitates continuous integration and continuous development [38]. A pipeline can be seen as a series of automated steps that a codebase undergoes. The flow of a pipeline can be defined within YAML, a data serialization language [39]. The YAML file is a series of instructions for the pipeline process [40] and can be divided into four phases [38].

The first phase is called the source phase, a stage which is crucial since it guarantees that every alteration/modification can be reversed or traced [41, 42, 43]. The source phase provides a contingency for developers in case something goes wrong. Then there is the second phase called the build phase [42, 41]. This is where the code that is retrieved from the source gets compiled into an executable product. The second phase ensures that code is properly constructed and it can contain a number of stages to create an environment that is suitable for further testing [42, 41]. After the build phase, it's time for the test phase [42, 41]. This is where the code is subjected to automated tests to thoroughly ensure that there are no bugs or issues [42, 41]. Lastly, there is the release phase, this is where the code gets deployed to the target environment, after passing all the tests, making it available to end users [44, 45, 41].

As aforementioned, a pipeline consists of four distinct phases. Within these phases tasks are used to break down the work into smaller pieces [46]. A task is a specific action or step in a pipeline, typically a script or an executable, that collectively makes up the entirety of a pipeline [46]. The tasks are executed in a certain order and each phase has its own set of tasks [46].

2.5.5 Azure Functions

Azure Functions are solutions provided by Microsoft which allows event based code execution without having to dedicate a persistent resource, what Microsoft refers to as a "serverless solution" [47].

There are two distinct models a function adheres to which dictates the execution environment [48]. The isolated worker model runs, as the name implies, in an isolated runtime environment thus avoiding version conflicts and grants configuration freedom [48, 49]. The other worker model is referred to as in-process and is executed in a shared runtime. This model requires less overhead and is therefore suitable for systems with a high volume of invocations [48, 50]. Users are not bound to specific resources and are instead charged based on the number of executions or the execution time [51], as detailed in section 2.5.5.3. This pricing model is particularly beneficial in scenarios where code needs to be run only sporadically. Microsoft refers to these events as "triggers", all which are bound to a specific Azure function [52]. Events that can trigger execution can take a number of shapes, such as receiving an HTTP request or timer-based invocation.

2.5.5.1 HTTP triggers

An HTTP Trigger Function activates when an HTTP POST request is sent to a URL which the function is listening to [1]. Listing 2.1 below displays the instantiating of an HTTP Trigger. Rows 1 through 3 are the function declaration while row 5 creates a logger file which is used for documentation purposes. Row 6 displays a message written to the log. Rows 8 through 12 show the process of creating and sending an HTTP response message to the senders.

```
1  [Function(nameof(HttpFunction))]
2  public static HttpResponseData Run([HttpTrigger(AuthorizationLevel.
       Anonymous, "get", "post", Route = null)] HttpRequestData req,
3      FunctionContext executionContext)
4  {
5      var logger = executionContext.GetLogger(nameof(HttpFunction));
6      logger.LogInformation("message logged");
7
8      var response = req.CreateResponse(HttpStatusCode.OK);
9      response.Headers.Add("Content-Type", "text/plain; charset=utf-8
       ");
10     response.WriteString("Welcome to .NET isolated worker !!");
11
12     return response;
13 }
```

Listing 2.1: Code skeleton for an isolated worker for HTTP Trigger [1]

2.5.5.2 Timer triggers

A timer trigger function is executed based on the passage of time [2]. This type of function is initiated with an argument that dictates the interval between function invocation. Listing 2.2 displays the syntax for a Timer Trigger. Row 2 through 4 is the function declaration setting the execution time to be every 5 minutes. Row 7 instantiates a logger file. Timer trigger functions does not require a return value since it is invoked by a timer.

```
1  //<docsnippet_fixed_delay_retry_example>
2  [Function(nameof(TimerFunction))]
3  [FixedDelayRetry(5, "00:00:10")]
4  public static void Run([TimerTrigger("0 */5 * * * *")] TimerInfo
       timerInfo,
5      FunctionContext context)
6  {
7      var logger = context.GetLogger(nameof(TimerFunction));
```

Listing 2.2: Code skeleton for a Timer Trigger example which is an isolated worker. [2]

2.5.5.3 Pricing

There are a number of pricing plans for Azure functions and is influenced by factors such as duration of commitment, performance and geographical area [51, 53]. Each resource and the subsequent pricing plan is tied to an Azure subscription, a cost center for billing purposes [54]. A pay-as-you-go consumption plan is ideal for smaller-scale functions [55, 54]. Under this plan, users receive 1 million free function executions or 400,000 GB-seconds of resource consumption. Billing depends on which limit is reached first: either 1 million executions or 400,000 GB-seconds. Once the free limit is exceeded, the charges are SEK 2.19 per million executions or SEK 0.000176 per GB-second [51, 53].

2.5.6 Azure table storage

An Azure table storage is an object database created for easy and cheap data storage. It is a NoSQL database meaning all sorts of documents can be stored in tables. Each document is referred to as an entity and includes several properties. The table storage does not require uniformity between entities [56, 57].

The properties within these entities can be of various data types, including `String`, `Boolean`, and `DateTime` [58]. Storage consumption for these data types varies: a `Boolean` [59] property occupies one byte, a `DateTime` property uses eight bytes [60]. A string's storage requirement is determined by its length, with each character taking up one byte plus an additional byte for the null terminator [61]. An Azure function can easily interact with a table storage through input and output bindings. Input bindings allow functions to read entities, while output bindings allow them to write to or delete from the table storage [52, 62].

Each Azure Table Storage is linked to a storage account. Billing is based on the amount of stored data and the number of operations performed. Costs also vary depending on the location of the table storage and the chosen data redundancy option [63]. For non-sensitive data, Local Redundant Storage (LRS) is an affordable choice, it replicates data three times within the same geographic area. LRS is not the most secure but is cost-effective for less critical data [64]. A storage account owner is billed SEK 0.4927 per GB stored, SEK 0.3558 per 10,000 write operations, SEK 0.0712 for 10,000 read operations while delete operations are free [63].

2.5.7 RESTful web API

Microsofts RESTful API builds upon the Representational State Transfer architecture, REST for short [65]. This design approach aims to create a decoupled, scalable and flexible systems through stateless client server architecture. This implies that messages with a payload enables the sender and an endpoint, message receiver, communicate and then act independently. The REST architecture additionally offers options to implement a system that supports documentation and thus traceability though Uniform Resource Identifiers, URI [66].

Microsofts RESTful API utilises the HTTP protocol over TLS when sending messages. Data in transit is therefor encrypted ensuring only the intended receiver can access it. This API also grants users a predefined number of HTTP methods which can be invoked, being POST, PUT, GET and DELETE [65]. The RESTful api implementation follows two major conventions that users need to adhere to, URL formatting and packet/ payload structure. The former is with regards to the API endpoint, the receiver, address and will only accept calls following the conventions set by Microsoft. The latter on the other hand refers to how the HTTP call header should be formatted all while the payload, the HTTP body, should follow a predetermined JSON scheme [67]. With the RESTful web API users can interact and control resources hosted in Azure [68, 69, 70].

3
Methods

The following chapter introduces and describes the use of Kanban as the primary work methodology in the project. This section starts by explaining how the workflow was structured. It then elaborates on the choice of Kanban as the work methodology and how it was used throughout the project.

3.1 Workflow

At the initial stage of the project, meetings with supervisors from New Wave Group were scheduled. This approach was taken to align the project's design with Chalmers' thesis guidelines and to ensure that the work met the objectives established by New Wave Group. Throughout the project, meetings were conducted on a weekly basis with New Wave Group, to discuss various problems and solutions associated with the task. These regular discussions ensured that the advancement made was agreed upon mutually and stayed in the correct direction, i.e. within the project scope. The project regularly set objectives to acquire knowledge or implementing features. Upon achieving these objectives, the outcome was evaluated and documented. To manage the development process, Azure Repos was employed together with a Git feature branch flow. The workspace was accessible to all project groups and stakeholders within New Wave Group. Development was done with Visual Studio, Azure Storage Emulator, Ngrok and Teams.

3.2 Use of Agile Practices

Kanban was chosen as the agile methodology for the project. This decision was based on a variety of factors. The primary reason was that it is a method used by New Wave Group. Secondarily, it is a methodology that does not have a preferred team size; it works well for both large and small teams [71, p. 67]. The foundation of the project is a component of a larger system, which made it important to be able to clarify the workflow for multiple involved stakeholders, both internal and external. Using a Kanban board visualised the flow and created clarity about what needs to be done, what was being done, and what had been done. Transparency was critical since the project work was of a cross-functional nature [71, p. 60–61].

Furthermore, Kanban contributed to creating structure by clearly and explicitly defining the meaning of the process. A commonly defined language laid the foundation for a standardised approach through set requirements and shaped reasonable expectations for a specific step in the development process [71, p. 62]. The method entails that the workflow and its boundaries are defined. These boundaries are, for example, the number of `Work-In-Progress` cards that can be in one stage on the Kanban board at the same time. The idea of defining an upper limit is to avoid a so-called "logjam", a bottleneck in the workflow. By setting a limitation, the developers were forced to continuously build upon the final product during the project [71, p. 65–66].

3.3 Cooperation with other subgroup

Coordination was necessary with another Bachelor of Science group working on a related part of the system. The other subgroup were responsible for creating templates of existing infrastructure. These templates were used in an Azure pipeline to either start or delete resource groups in the could. It is important to clarify that the project was divided into two parts, each handled by different groups. The overall solution can only be fully realized when both parts are integrated and some cooperation was therefore needed in the final stages. The work overlapped between the groups during the creation of the pipelines, since the code needed to format request bodies after a specific pipeline structure. This project specifically focused on creating an application to trigger pipelines using templates created by the other subgroup.

4

Implementation

This chapter begins with an overview of the finalized product and then outlines the system design. Then, it presents the business logic in detail, followed by a description of the testing, and finally discusses the projected costs of the system.

4.1 Product Overview

The finalized product is an automated system that manages Azure test environment instances directly from Teams. Users can interact with the system using three different command types; query, direct and automation. Commands are written in Teams, which are sent via an outgoing webhook and processed by an HTTP trigger. The direct commands triggers a pipeline whilst the automation command stores it for later execution. Entering a query command on the other hand displays a text in the Teams chat with information that aids the user.

4.1.1 Input formatting

User input consists of a set of string values in a precise sequential order. The order is crucial because it allows values to be parsed in a standardised way. Deviating from this structure renders the command invalid. The structure of the command is predefined based on the parsing logic, however the strings that compose a command are easy to modify. All commands and flags and are defined as constants within a C# class. Constants make it convenient to change the commands in the code. The common ground for each command is that they are initiated with the same keyword `@webhook`. This keyword is essential since it is considered an entry point of the program and prompts Teams to send off the message to the specified endpoint for the outgoing webhook.

As mentioned in Section 4.1, a query command aims to assist with the interaction. It starts with the same keywords as aforementioned, i.e. `@webhook`, followed by a query. There are four commands within this group. The `help` query which tells the user how to interact with the system, `resources` that lists all supported resources, `scheduledtasks` which lists all scheduled tasks and lastly `examples` which displays examples. Figure 4.1 shows the `help` command which lists all actions that can be taken in the system, and acts as a guide to the user.

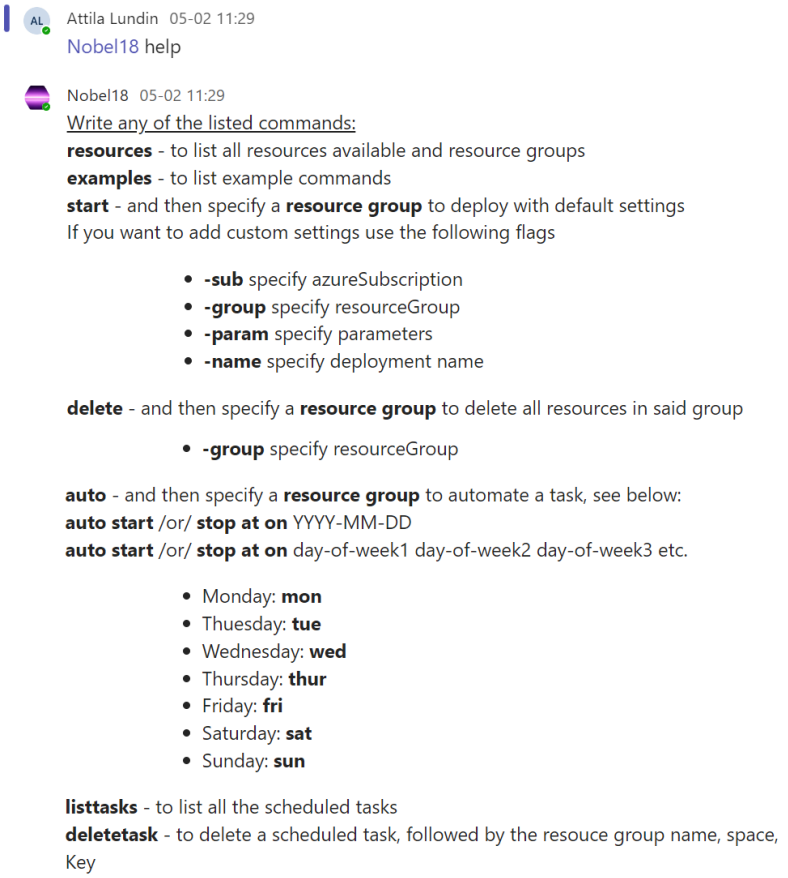

Figure 4.1: Response from the application using the help command

Direct commands starts with an action, for instance `start` or `delete`, followed by a resource. Additional parameters and flags are used when more specific actions are needed. The system currently supports four flags. The `-sub` flag enables the user to specify which Azure subscription the resource will be linked to while `-rg` specifies the group resources a resource will run in. The `-param` flag points to a file containing settings and, as the name indicates, parameters. Lastly the user can specify a name the process will be executed under using `-name`. One or all the flags can be individually specified. Figure 4.2 shows an example of a direct command that starts a resource with a flag set.

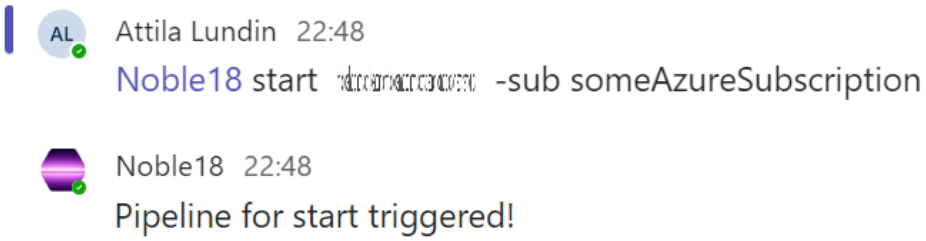

Figure 4.2: Start of a resource with the subscription flag set

Automation, apart from direct commands, are more complex as they involve scheduling. The initial structure of automation commands are identical to direct commands. However, to differentiate, all the automation commands include an automation keyword and is then followed by a resource. The user then needs to specify an action, such as start or delete after which the time of execution is specified. A user can either specify a date or day. The former will create a task that executes once while the latter will execute the task each weekday. A user can specify multiple actions and dates in one message. Figure 4.3 is an example of the automation command.

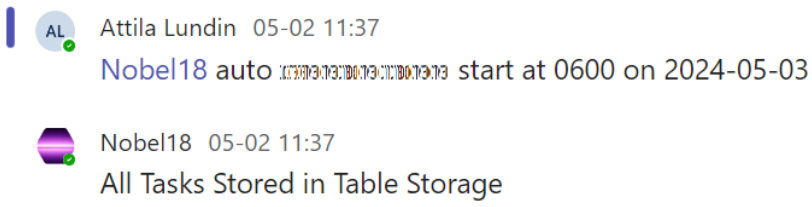

Figure 4.3: Response from using the automation command, scheduling a resource group to start once

A user can also remove scheduled tasks. This action typically requires a query to identify the unique key of the task by issuing `listtasks` command. Once the key associated with the task to be removed is identified, the user can issue the command to delete the task. Removing a task is done by specifying the `deletetask` command accompanied by the resource and the key as displayed in Figure 4.4.

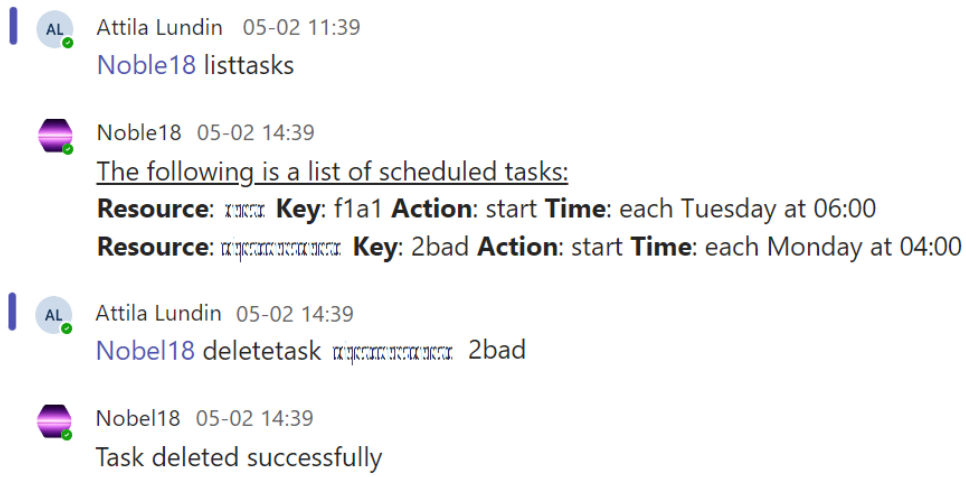

Figure 4.4: Response from using the list tasks command and then deleting a scheduled task

As seen in Figure 4.5 an incorrect command is issued. The system then proceeds to prompt a default message back to the user, indicating faulty formatting of the command. This measure providing guidance on what users should do next.

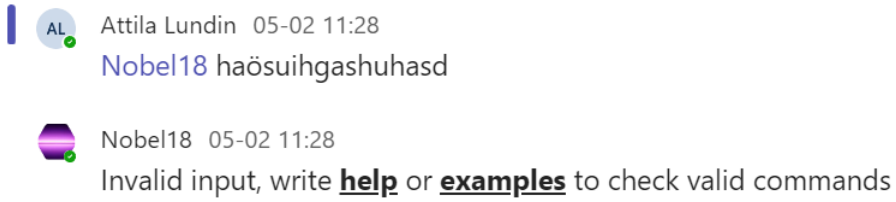

Figure 4.5: Response from invalid user input

4.2 Design

The project and solution were created based on requirements from New Wave Group. This chapter highlights and clarifies the main influences that shaped the solution and the overall design.

Several tools were considered during the designing of the system. Microsoft, for example, offers out-of-the box solutions which are easy to learn, use and interact with. Prebuilt tools and systems were ultimately disregarded since they came with fixed monthly cost or other paywalls. Tools from untrusted sources, i.e. not Microsoft, were excluded from the project since security was a major consideration. Steps have been taken to mitigate the risks and make the codebase as secure as possible. All data in transit needs to be sufficiently encrypted, the system thus exclusively uses HTTPS and vetted authorization methods. Sensitive information such as API keys are stored as environment variables. Additionally the solution has support for more secure storage services provided by Azure such as Azure Key Vault.

Cost was one of, if not the biggest factor of influence in the design. Added complexity in user interaction and the codebase was preferred over comparatively larger billing. A solution that follows a pay as you go plan, i.e. consists of pure variable cost depending on utilisation, was highly favoured. This resulted in a system architecture that was carefully designed to balance simplicity and cost-effectiveness. The system, at its core, is designed to issue commands in Microsoft Teams that acts as an extension and simplification of Azure CLI operation. Figure 4.6 illustrates the entire life cycle of a command.

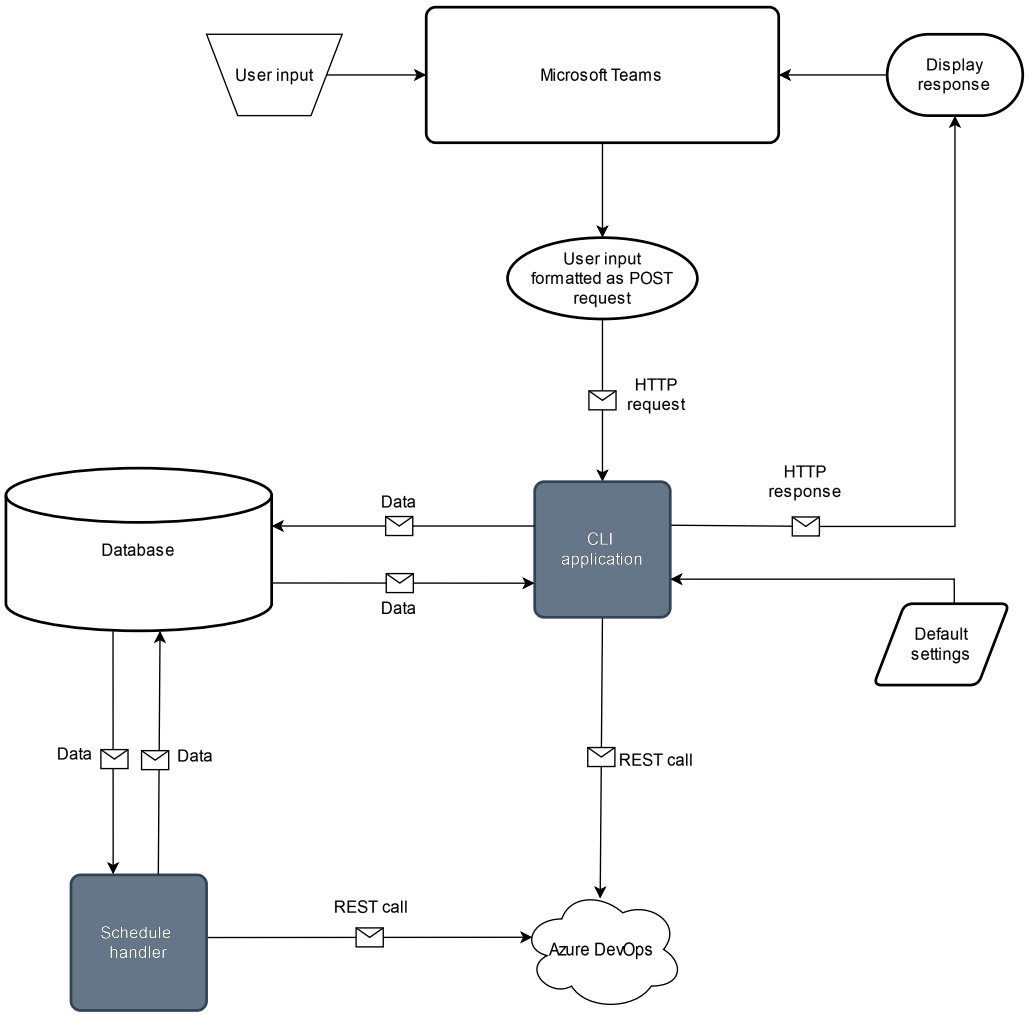

Figure 4.6: Design overview, note that process marked as blue are custom defined system components

Figure 4.6 illustrates that user input is transmitted to the `CLI application` via a POST request. The process involves establishing an outgoing webhook within a Microsoft Teams team. To achieve this, one navigates to the team's app section and selects the option to create a new outgoing webhook. The necessary information, such as the webhook name, endpoint URL, and a description, is then specified. The endpoint URL should be set to where the `CLI application` is hosted. Upon creation, an HMAC security token is generated and displayed. This token must be configured as an environment variable in the `CLI application` to facilitate authentication.

When a command is entered by a user, it is instantly processed by the `CLI application`. Essentially, a command is transmitted via the outgoing webhook from Teams, generating a POST request that carries the command as payload to the function's URL. The `CLI application` is an azure function triggered by HTTP calls and contains all the logic to process the commands and respond to the user. It additionally supports data transmission to and from a `table storage` database. The `Schedule handler` corresponds to an Azure timer function and focuses solely on polling the database and updating entries. Both Azure functions have the ability to send a request to a specified endpoint to run a pipeline. The pipelines can either start or delete a resource. Each entry in the `table storage` database can be seen as a task for the `Schedule handler`. The tasks contain all the necessary data to trigger a pipeline after a specified time has elapsed.

The application has a settings file which needs to be considered when maintaining the system. This file contains settings for each resource group and is structured in a JSON format. The root object within the files contains objects for each resource group. The nested objects in turn contain all the different settings for both start and deletion. The reason for this structure is to make it easy to add and update settings for resource groups during runtime without modifying any logic. The JSON file structure is displayed in Listing 4.1.

```
{
  "genericapi": {
    "start": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
      "parameters": "string/path/to/parameterfile.
        bicepparam",
      "name": "string-ProcessName"
    },
    "delete": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
    }
  },
  "genericshop": {
  "start": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
      "parameters": "string/path/to/parameterfile.
        bicepparam",
      "name": "string-ProcessName",
      "storageAccName": "string-StorageAccountName"
    },
    "delete": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
    }
  },
  "genericcms": {
        "start": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
      "parameters": "string/path/to/parameterfile.
        bicepparam",
      "name": "string-ProcessName"
    },
    "delete": {
      "azureSubscription": "string-AzureSubscription",
      "resourceGroup": "string-SelectedResourceGroup",
    }
  }
}
```

Listing 4.1: JSON file for the default settings, where each object denotes a resource which is supported by the system

4.3 Table storage entities

Each entity in the `table storage` needs to follow a specified formatting. Figure 4.7 displays the six columns each valid entry is expected to have, together with an example entry.

PartitionKey	RowKey	Timestamp	IsStart	IsPersistant	ScheduledTime
exampleResource	p8er	2024-05-25T17:22:50.5175457Z	true	false	2024-05-31T06:00:00.0000000Z

Figure 4.7: Table Storage Columns

The `PartitionKey` column should contain a string value specifying the resource group for the task. The `RowKey` is a four-character string, combining numbers and letters, that serves as a unique identifier for each task. The `Timestamp` column records the creation time of the task, while the `ScheduledTime` column details the time and date set for execution. Both of these columns use the DateTime data type. `IsStart` and `IsPersistent` are boolean flags; `IsStart` indicates whether the task is to start or delete the resource, and `IsPersistent` determines whether the task is recurring or should be executed only once before deletion.

4.4 RESTful request body formatting

RESTful request bodies need to be formatted in a specific and precise way in order for Azure to trigger a pipeline run. The formatting is dictated by what parameters are defined in the pipeline template configuration. If a defined parameter is absent from the request body, the API will not trigger a pipeline run. Although values in the request body that do not have a corresponding parameter will be disregarded. The pipeline that instantiates resources within a resource group has parameters as seen in Table 4.8.

```
parameters:

- name: azureSubscription
  displayName: 'Azure Subscription'
  type: string
  default: "we-ds-dev-studentlab"

- name: resourceGroup
  displayName: 'Resource Group'
  type: string
  default: "we-ds-dev-studentlab"

- name: parameters
  displayName: 'Bicepparam File'
  type: string
  default: "we-ds-dev-microshop/cms.bicepparam"

- name: name
  displayName: 'Deployment Name'
  type: string
  default: "testDeployCms"

- name: storageAccName
  displayName: 'Storage Acc Name'
  type: string
  default: "stdntlabdummystrgacc001"
```

Figure 4.8: Deploy parameters for a pipeline that starts a resource group

The request body sent to the API endpoint for a pipeline that starts a resource group is displayed in Listing 4.2. As displayed below, the JSON file contains an object with the name `templateParameter`. The object will be checked by the API for key-value pairs matching the defined parameters in the pipeline template configuration. This object must contain each pipeline parameter and have the same name and data type.

```
{
    "templateParameters": {
        "azureSubscription": "someStringValue",
        "resourceGroup": "someStringValue",
        "parameters": "someStringValue",
        "name": "someStringValue",
        "storageAccName": "someStringValue"
    }
}
```

Listing 4.2: JSON body for a request that triggers a pipeline that starts a resource group

The pipeline that removes a resource group only contains the first two elements in the `templateParameters` object, i.e. `azureSubscription` and `resourceGroup`. This is displayed in Figure 4.9 below.

```
parameters:

- name: azureSubscription
  displayName: 'Azure Subscription'
  type: string
  default: "we-ds-dev-studentlab"

- name: resourceGroup
  displayName: 'Resource Group'
  type: string
  default: "we-ds-dev-studentlab"
```

Figure 4.9: Deploy parameters for a pipeline that deletes a resource group

The body structure is identical to the request which starts a resource group but only needs the `azureSubscription` and `resourceGroup` variables. Listing 4.3 below describes the JSON structure.

```
{
    "templateParameters": {
        "azureSubscription": "someStringValue",
        "resourceGroup": "someStringValue",
    }
}
```

Listing 4.3: JSON body for a request that triggers a pipeline that deletes a resource group

4.5 CLI application

The main function of the `CLI application` is `Run` and executes when an POST request is received to an endpoint. Figure 4.10 describes the flow of the Run function.

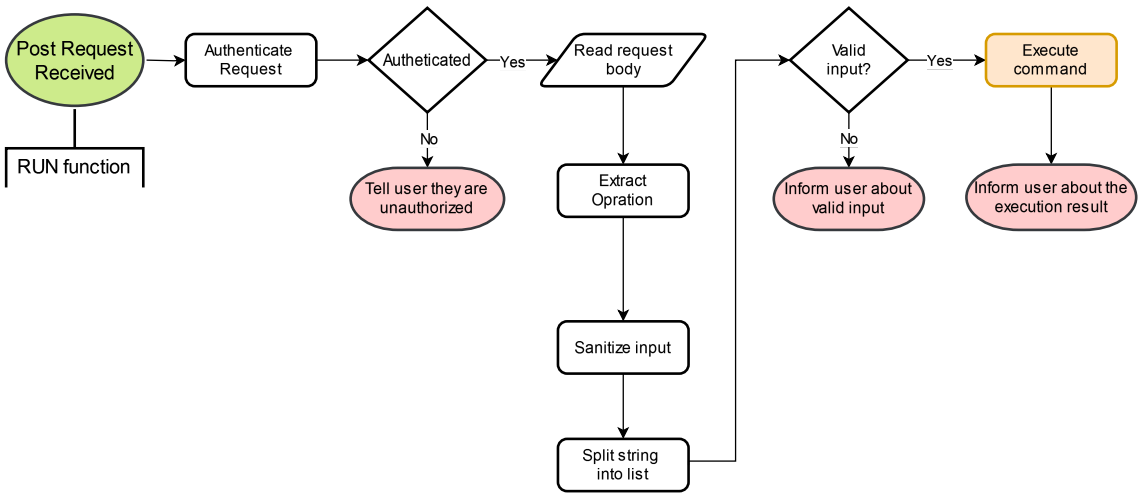

Figure 4.10: Flowchart over the Run function. Each process filled with a colour has a separate flowchart further explaining the logic

Initially the request is authorised. Then the function hashes the request using an authentication token shared between the outgoing webhook and the system. The calculated hash is then compared with a value in the request header. If the values match the origin is guaranteed to be the webhook. If, on the other hand, the values do not match, a response with HTTP status code for unauthorised is returned to the sender.

Following a successful authentication the system will read the request body, parse the content and extract the user-defined operation. Incorrect commands will return a message informing the user of permissible inputs. The parsing removes unwanted characters and formatting such as HTML-tags. The user input is then split into a list of words delimited by space. A check is conducted on the first element in the list, the contents must match one of the predefined supported operations. If no match is found, a corresponding error message is prompted back to the user. After parsing the system will have access to an operation and a list containing the rest of the user input.

In accordance with Figure 4.10 the system then invokes a function that executes the command. The flow of command execution is described by Figure 4.11.

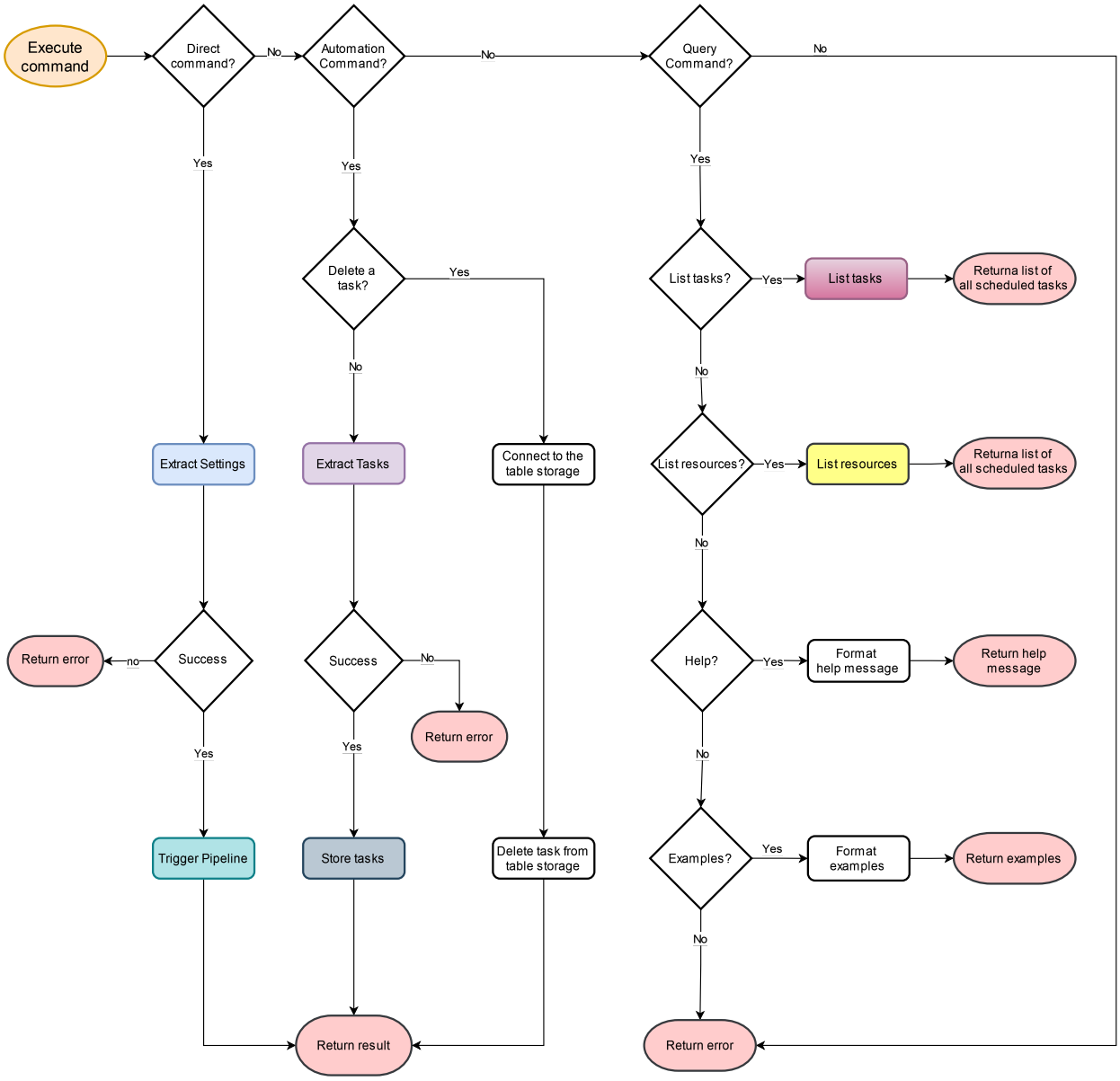

Figure 4.11: Flowchart over the execute command function.

The `Execute command` function requires three inputs: the previously extracted command, a logger to record noteworthy events, and a list containing user inputs. The function primarily consists of a switch statement that determines the flow based on the command's category, as illustrated in Figure 4.11. Subsequent sections will detail the execution of each command type according to Figure 4.11 and offer a thorough explanation of the remaining steps in the command's life cycle.

4.5.1 Direct command

A direct command can be issued to manage resource groups, specifically, it should either start or delete a resource group. However, it's important to note that the actual creation or removal of a resource group is not performed by the `CLI application` itself. Instead, these actions are carried out through two separate Azure pipelines. Consequently, the system's role is simply to initiate these pipelines. Regardless of whether a resource group is being started or deleted, the command flow and the function structure remains the same; the only variation is the specific payload transmitted with the RESTful API call and the API endpoint. Figure 4.12 below demonstrates how the system starts the process of formatting the aforementioned payload.

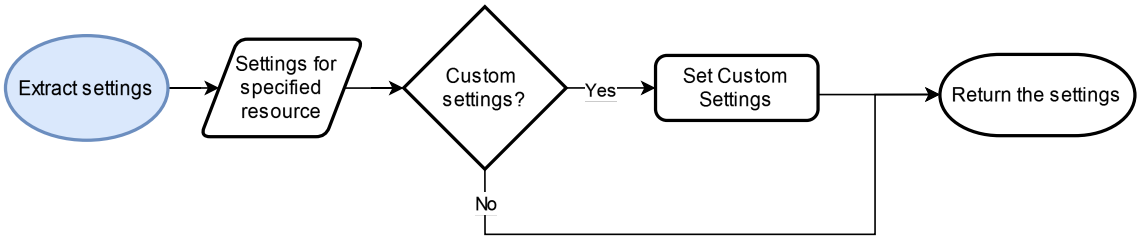

Figure 4.12: Flowchart over the extract settings function

The process begins by loading a JSON file that contains all the default settings. Next, it tries to fetch the settings for the user specified resource group. If the fetch is successful, an object is created with all the default settings. The system then checks the user input for any custom parameters. If custom parameters are found in the list of user input, they override the default settings. If the settings cannot be successfully extracted, a response is sent back to the user, informing them of the issue and providing guidance on how to compose a valid command.

When the function terminates successfully, it returns an object that contains all the settings. After which another function is invoked which triggers an Azure pipeline. The process is described in Figure 4.13.

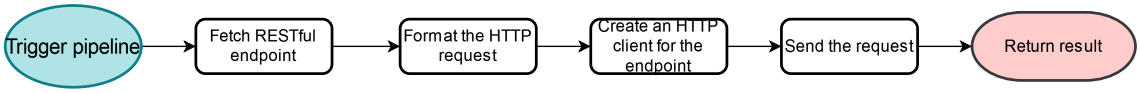

Figure 4.13: Flowchart over the trigger pipeline function

First, the system retrieves a URL from the environment variables corresponding to the action type, i.e. starting or deleting a resource group. Then, it formats a request with settings as the body. To send this request, a HTTP client is created, configured with a personal access token. The client then sends the request to the specified URL endpoint. The system subsequently captures, logs and returns the response message and status.

4.5.2 Automation command

The process of executing an automation command is initiated by extracting tasks from the user input. This is done by invoking a function which is illustrated in Figure 4.14.

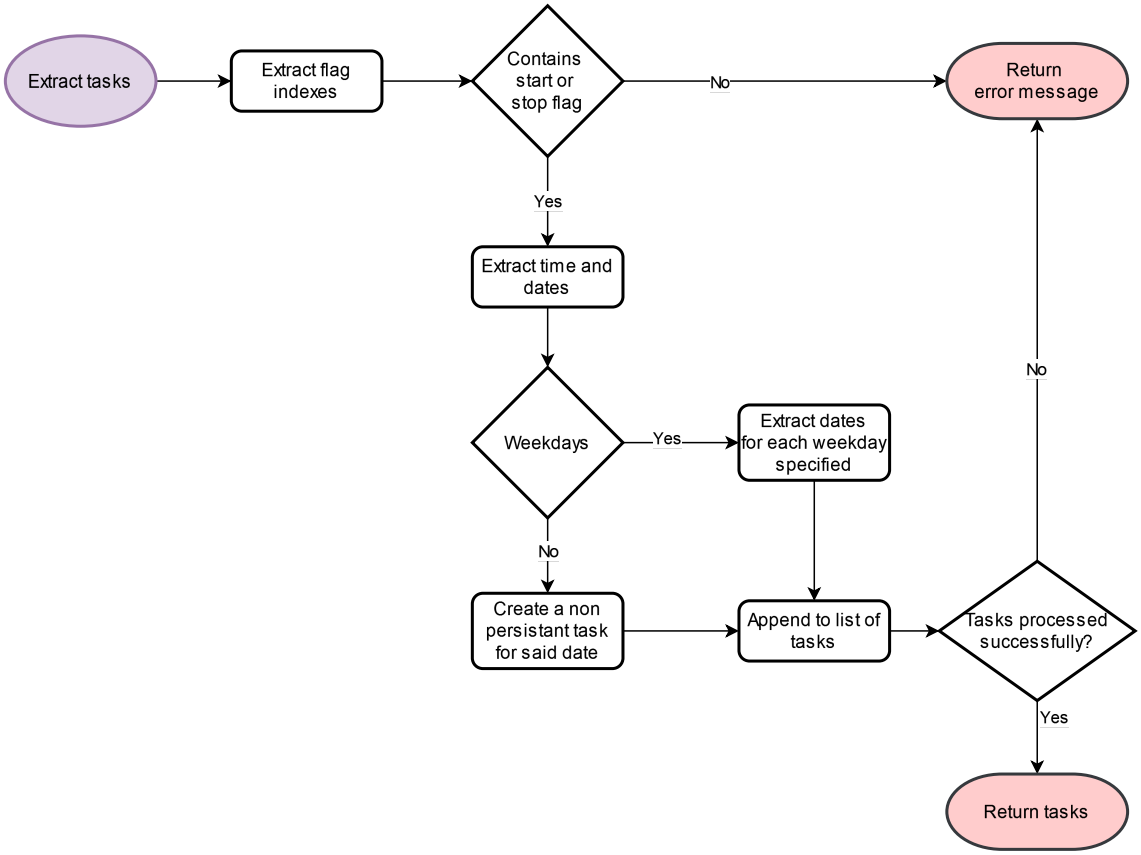

Figure 4.14: Flowchart over the extract tasks function

The `Extract tasks` function processes a list of parameters, each consisting of key-value pairs that include action, time, and date flags. It first determines the position of each keyword to extract the values specified by the user. Task extraction occurs in two phases: first, the system processes the user input to schedule the start, and then it moves on to schedule the deletion of resource groups.

Dates and times are extracted first. If a specific date is provided, a task object is created with said date and time. The action is set together with a flag value for non-persistence, meaning it will execute only once. If the user input is accompanied by one or more weekdays, the function calculates the date for the next occurrence of each specified weekday. A separate task is then created for each of these days at the designated time. In these cases, the action type is set together with a flag denoting the task as persistent. A persistent task recurs indefinitely until the task is manually removed. Each task identified from the user input is then added to a separate list that holds all tasks.

If the list of tasks ends up empty, the function returns a boolean value of false along with an error message indicating that no tasks were found. Conversely, if tasks are present, it returns a boolean value of true and the list of all tasks.
When tasks are successfully created, they are stored in a `table storage` by establishing a database connection and inserting each object as a table entity, depicted in Figure 4.15.

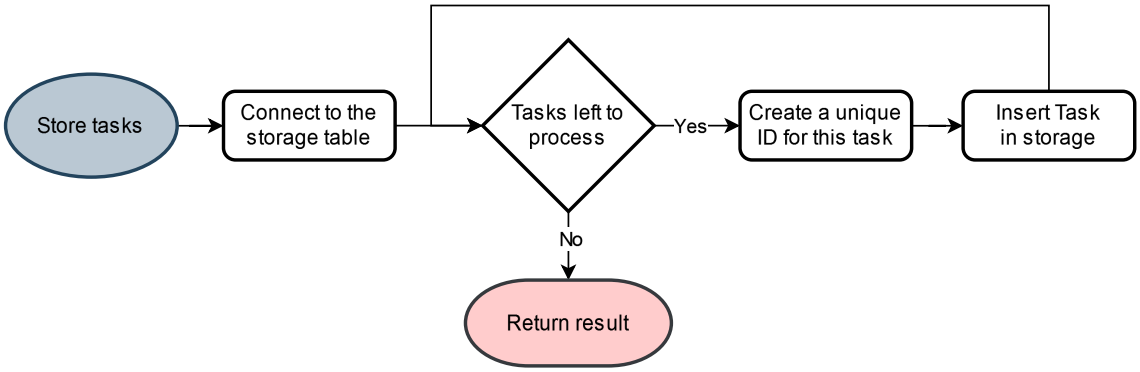

Figure 4.15: Flowchart over the store tasks function

The process involves creating a Global Unique Identifier, GUID from which only the first four characters are used. Given that the database will contain only a small number of entries, the likelihood of a collision is minimal. However, if a collision does occur, the process is repeated and a new GUID is generated to serve as the entity's row key. If there are no tasks left to process, the system returns the result in a message.

As mentioned before, a persistent task needs to be removed manually. The removal a task can bee seen in Figure 4.11. When the automation task is deleted, the system proceeds to create a connection to the `table storage` if the command is to delete a task. The list containing user input is used to identify the element to delete. The first element in the list will be assigned as the partition key and the second as the row key. The function then attempts to delete this entry. The result of the execution is returned, even if the deletion fails. This is so that the outcome will be displayed in the team's chat.

4.5.3 Query command

As mentioned before in section 4.2 input formatting, query commands can be issued by the user to aid in interaction with the CLI. Each query will prompt the system to return a formatted string that displays relevant information. If the command is for help, a static message is returned. The message uses commands and flags defined within the constants class. This keeps the output up to date even if keywords are changed. The same principle is applied for the example command which provides the user with concrete examples of valid inputs.

The logic differs when the user sends a command to list available resource groups. Executing this command invokes a function that reads a JSON file containing all default settings. By using the JSON file with the default settings the available resource groups will stay up to date even if the settings file is modified. The system prepares a string containing the name of each element in the JSON file. See the Figure 4.16.

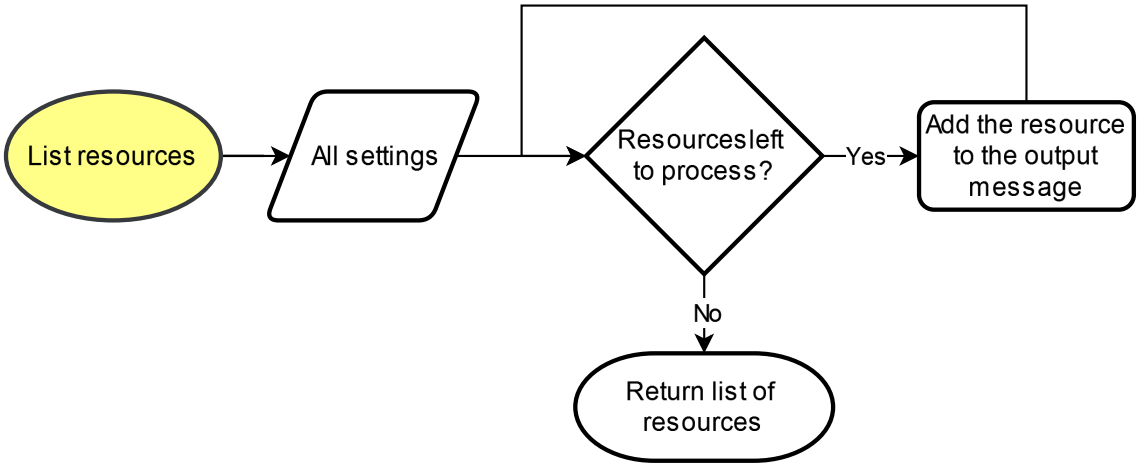

Figure 4.16: Flowchart over the list resources function

Listing the stored tasks works in a similar way as listing resource groups. The function connects to the `table storage` and retrieves all entries. These entries are then iterated over and added to the output message. The message will thus contain a list of all the scheduled tasks with its relevant properties such as time, date or day and the action type. The flag denoting if a task is persistent determines whether to display a date or a day. The former applies for non-persistent tasks while latter is for persistent tasks. See Figure 4.17.

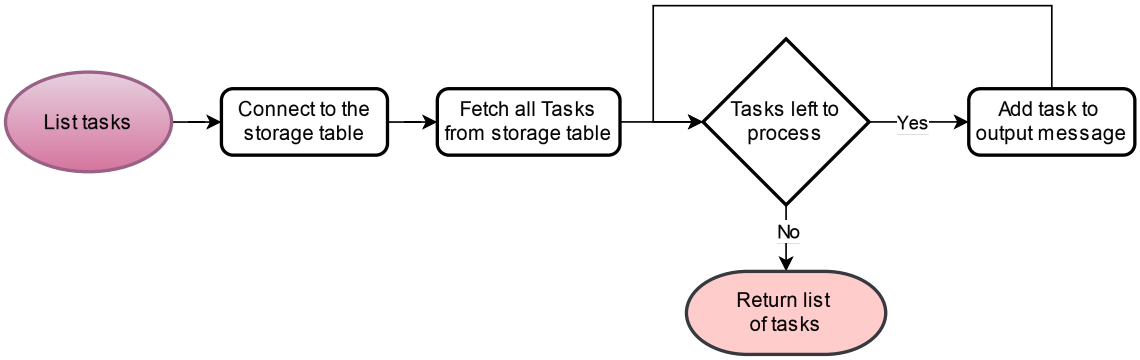

Figure 4.17: Flowchart over the list tasks function

4.6 Schedule handler

The `Schedule handler` periodically reads from a storage to determine whether it is time to start or stop a resource. The flowchart depicted in Figure 4.18 describes the execution process of scheduled tasks.

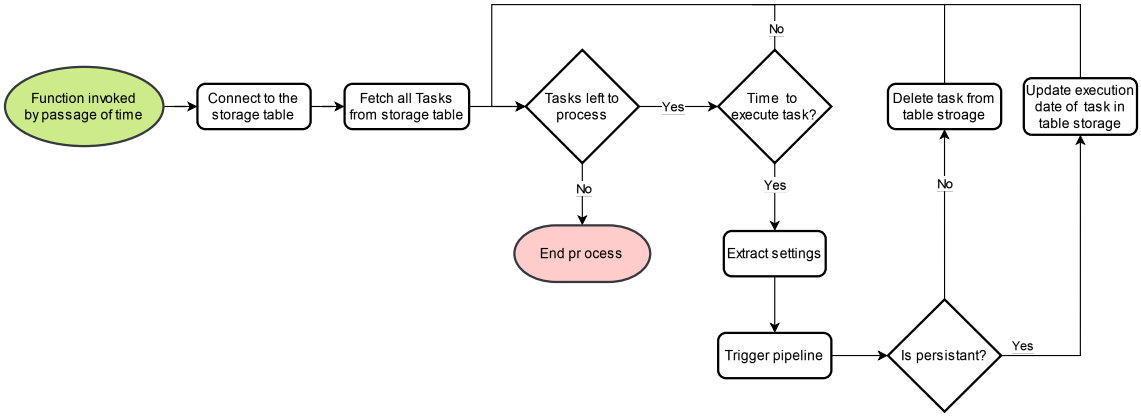

Figure 4.18: Flowchart over the schedule handler

Initially the `Schedule handler` activates a function that connects to `table storage` to retrieve all stored tasks. Each task is then individually assessed to determine if the conditions for execution have been met. This involves comparing the value in the `scheduledTime` variable with the current time. The conditions for execution is met if the current time and date aligns or is after the `scheduledTime` value.

Conversely, if any task meets the conditions for execution, the system proceeds to extract default settings for said task. A POST request is then formatted, just like in the `CLI application`, with the extracted settings. The request is then used to trigger a pipeline. Once the task is executed, it is reassessed to determine if it is persistent. If the task is persistent, its timing is updated in the schedule; otherwise, it is deleted from the `table storage`. This ensures that only relevant tasks remain scheduled. When all tasks are processed the `Schedule handler` will terminate and sleep until it is next invoked.

The `scheduledTime` is not necessary for the `CLI application` to work. New Wave Group can decide to only start the `CLI application`, negating starting processes for the `Schedule handler` and Azure `table storage` only keeping functionality for direct and query commands.

4.7 Testing

Throughout the project, the two main testing methods employed were user and integration testing.

4.7.1 User testing

Ngrok was used to redirect packets from the Teams webhook to localhost. This setup allowed for local testing of the application, which was necessary because webhooks require external hosting. Ngrok creates a public URL which forwards packets to a specific port on the local machine. Figure 4.19 demonstrates the start command for Ngrok forwarding packages to `localhost:7199`.

Figure 4.19: Command to start ngrok

When Ngrok is started, the terminal will display data as shown in figure 4.20. The URL displayed under `Forwarding` is used as the Azure function endpoint in Microsoft Teams. Each time Ngrok is started a new URL is created which routes packets to the previously specified port.

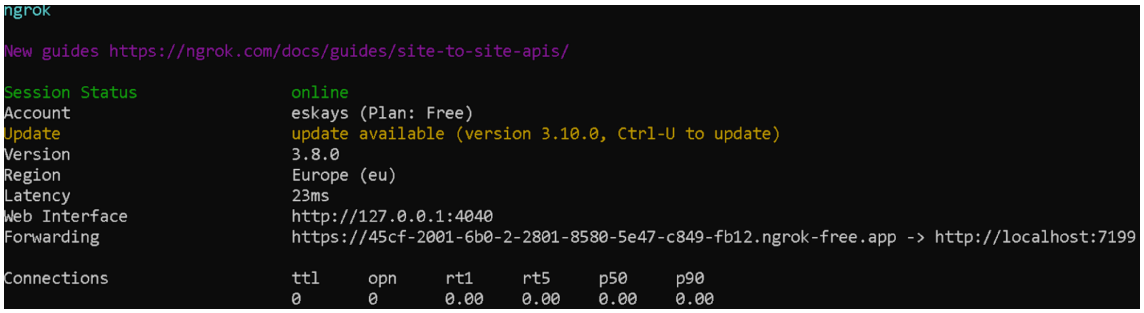

Figure 4.20: Terminal output of the Ngrok process

User testing was conducted through creating various scenarios where the developers typed in commands directly in Teams to check if it matched the expected outcome. These scenarios covered a range of actions, such as initiating environments, querying the system, and scheduling tasks.

4.7.2 Integration testing

Integration tests were also used to ensure that the interplay between components worked as intended. The testing was conducted on the interaction between the Teams webhook, Azure functions, `table storage` and Azure services. This process was done incrementally as new functions were integrated to maintain a working system. Subsequent tests were conducted on the interaction between the `CLI application` and the Azure cloud via API calls. These included sending data through POST requests to activate a pipeline designed to assign variables based on the values received from the request.

After each component was tested carefully, the last phase started. This involved testing the system altogether. A sandbox environment, essentially a resource group, was created to further test the application. This confined environment allows testing to be conducted without affecting the actual applications of New Wave Group. The test was built upon creating POST requests to trigger pipelines that initiated resources into the sandbox resource group. This test encapsulated the entirety of the project, from initiation of command to deploying resources in Azure. A successful result indicated that the application was implemented correctly.

4.8 Projected system cost

There are two primary cost-driving variables: the total number of Azure Function executions between the `CLI application` and the `Schedule handler`, and the data stored in- and total operations on the `table storage`. The Azure functions operate on a pay-as-you-go consumption plan, and the `table storage` employs LRS redundancy, both situated in Western Europe. Consequently, the total cost of the system hinges solely on the marginal costs associated with these variables. TC represents the total cost, MC_{AF} is the cost of Azure functions, and MC_{TS} is the cost of the `table storage`

$$TC \;=\; MC_{AF} \;+\; MC_{TS}$$

The cost for Azure functions is determined by resource consumption or function executions beyond the free grant. The implemented functions both consume under 400,000 GB-seconds per one million executions. Therefore, the marginal cost for Azure functions MC_{AF} is based on any executions over the free grant.

The responsiveness of the `Schedule handler` depends on the polling interval. The number of monthly executions can be calculated using the formula below. To maintain a conservative approach in cost analysis, each month is said to contain 31 days.

$$\frac{N_{exec}}{h} \;\times\; 24\frac{h}{d} \;\times\; 31\frac{d}{M} \;= \frac{N_{exec}}{M}$$

N_{exec} is the number of executions while M is months, h is hours and d is days. A responsive `Schedule handler` executes tasks as precisely as possible, managing tasks on a minute-by-minute basis. This results in 60 function executions per hour.

$$\frac{60\ executions}{h} \;\times\; 24\frac{h}{d} \;\times\; 31\frac{d}{M} \;= \frac{44,640\ executions}{M}$$

Each command issued by a user corresponds to one function execution. Given the frequency at which the `Schedule handler` operates, the system can accommodate up to 955,360 user commands per month without surpassing the limits of the free tier. With access restricted to only a select few users, this setup enables virtually unrestricted interaction. The formula for the total cost can thus be simplified.

$$TC \;=\; MC_{TS}$$

The total expenditure is therefore solely dependent on the marginal cost for the `table storage`. The costs for table storage are divided into three components: the cost for data storage, the cost for write operations, and the cost for read operations.

To estimate the data storage cost, denoted as C_{data}, the size of each entry in the database can be calculated. An entry includes two `Strings`, two `DateTime`, and two `Booleans` properties. The `PartitionKey` string is assumed to have a maximum of 40 characters, resulting in a storage size of 41 bytes, i.e. 40 characters plus a null terminator. The second string, with four characters and a null terminator, occupies 5 bytes. Each `DateTime` property uses 8 bytes while `Booleans` uses 1 byte. The total size for each entry is therefore estimated to be 64 bytes.

$$Storage\ cost\ =\ C_{data} = \frac{0.4927 \times 64}{10^9} SEK$$

The cost of read and write operations in the system depends on whether a task is persistent. A persistent task requires one read and one write operation for each `Schedule handler` execution. In contrast, a non-persistent task requires one read and one delete operation. Since delete operations are free, persistent tasks cost more and are therefore considered in the projections. The cost per persistent task can be expressed as follows: C_r represents cost of read operations, and C_w represents the cost of write operations.

$$Execution\ cost\ =\ C_r\ +\ C_w\ = \frac{0.0712\ +\ 0.3558}{10,000} SEK$$

The marginal cost for `table storage` can be determined by multiplying the total number of monthly executions by the number of tasks and adding the initial write operation to store said task and the storage cost for all tasks.

$$TC = (Storage\ cost + Cost\ to\ add\ Task + Montly\ execution\ cost) Number\ of\ tasks$$

$$TC\ =\ (C_{data} + C_w + (C_r + C_w)\frac{N_{exec}}{M}) N_{task}$$

5
Results

This chapter will provide a projected cost of the system using different configurations, additionally the potential savings will be reviewed.

5.1 Projected net cost

Three possible configurations were considered to project the net cost of the system. The first is a responsive system where the `Schedule handler` polls the `table storage` every minute. The other configuration is more realistic, only polling the `table storage` once every 30 minutes. The third configuration, on the other hand, does not initiate the `Schedule handler` or the `table storage`. The latter only supports queries and direct commands.

Using the number of executions and the expression for total cost derived in section 4.8, a monthly projection can be created for a responsive system, denoted as TC_{60}.

$$TC_{60} = (\frac{SEK\ 0.4927 \times 64}{10^9} + \frac{SEK\ 0.3558}{10,000} + (\frac{SEK\ 0.0712\ +\ 0.3558}{10,000}) \times 44,640) N_{task}$$

The same principle can be applied in order to project the monthly cost for the realistic configuration, denoted as TC_2. This is done by calculating the total monthly executions then using the expression from section 4.8.

$$\frac{2\ executions}{h}\ \times\ 24\frac{h}{d}\ \times\ 31\frac{d}{M}\ = \frac{1,488\ executions}{M}$$

$$TC_2 = (\frac{SEK\ 0.4927 \times 64}{10^9} + \frac{SEK\ 0.3558}{10,000} + (\frac{SEK\ 0.0712\ +\ 0.3558}{10,000}) \times 1,488) N_{task}$$

Based on the expressions TC_{60} and TC_2 a graph displaying the cost based on the number of tasks is shown in Figure 5.1.

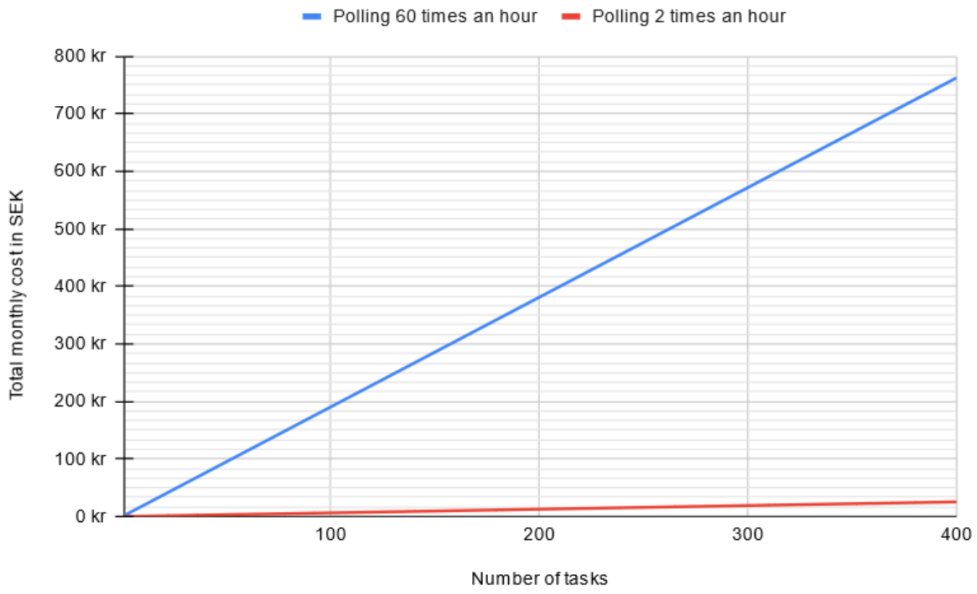

Figure 5.1: Total cost based on the number of stored tasks, for a system Schedule handler that polls the system 60 times and 2 times an hour.

The cost of a responsive configuration with 400 tasks amounts to SEK 762.46, whereas a more realistic configuration with the same number of tasks costs SEK 25.43. The third configuration, i.e. no support for scheduling, can easily be configured by choosing to not run the `Schedule handler` and the `table storage`. The only billable component is the `CLI application` which is free.
The cost of each resource group needs to be taken into consideration to further contrast the potential savings. Figure 5.2 through 5.4 display the costs for each implemented resource group.

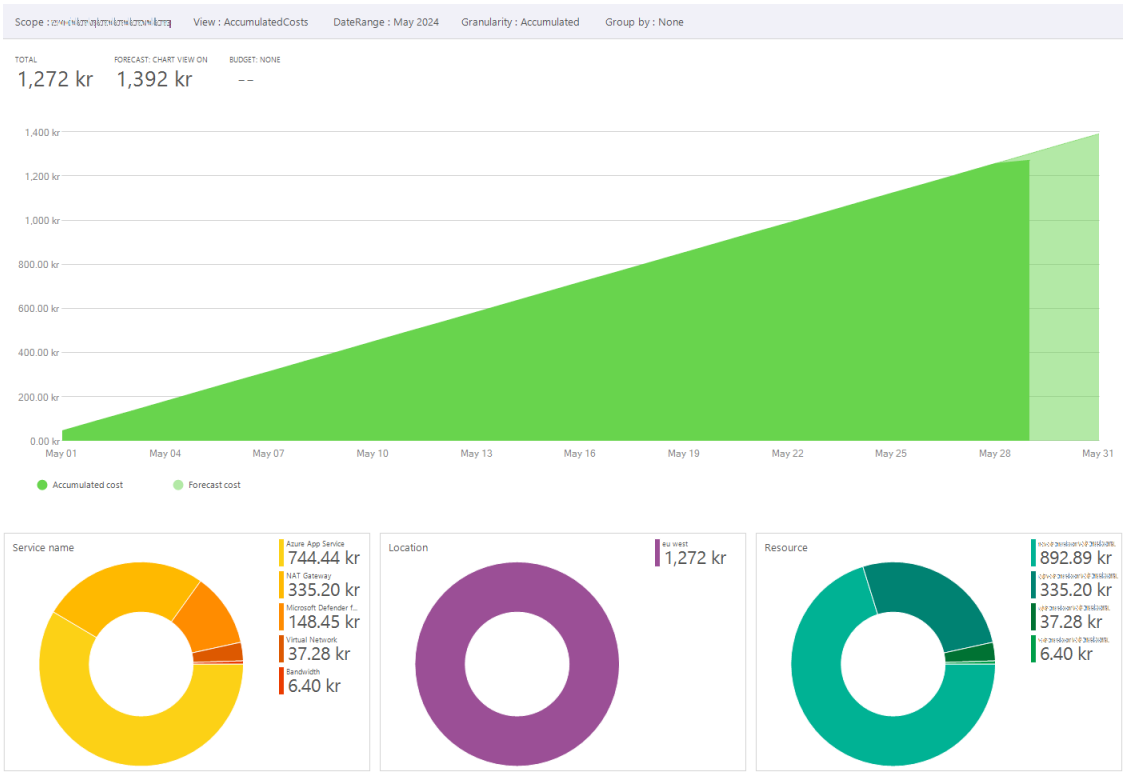

Figure 5.2: Monthly cost for genericCms

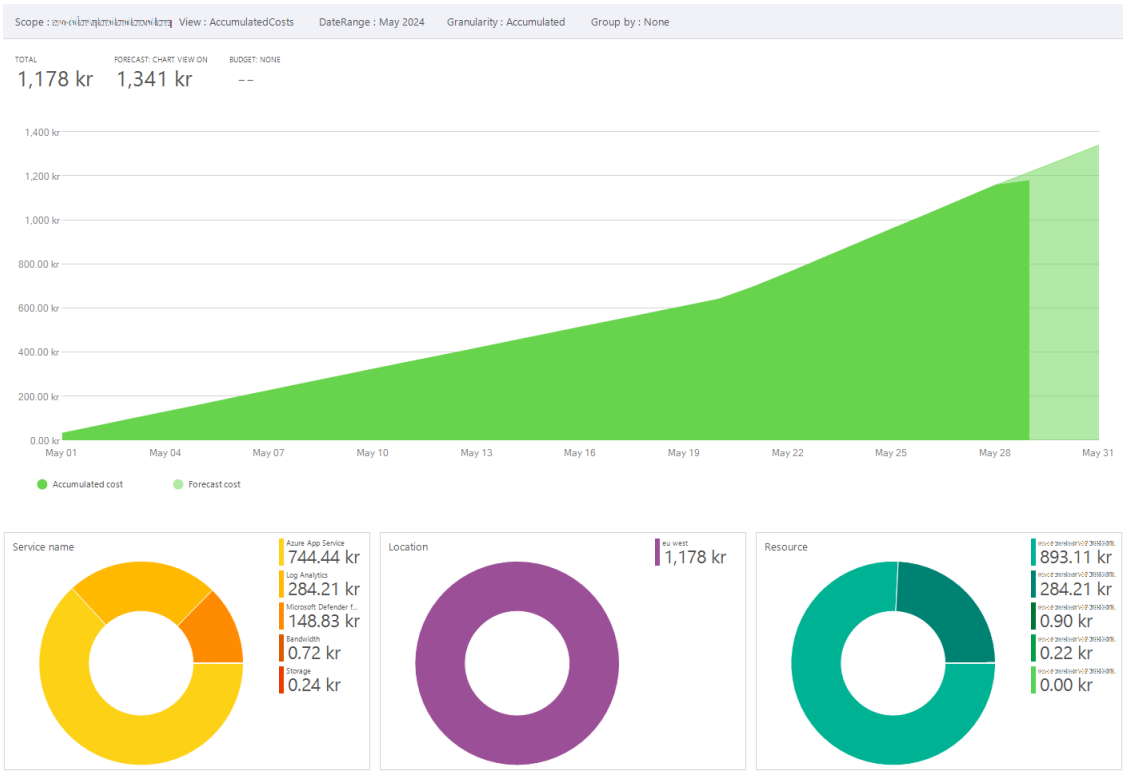

Figure 5.3: Monthly cost for genericAPI

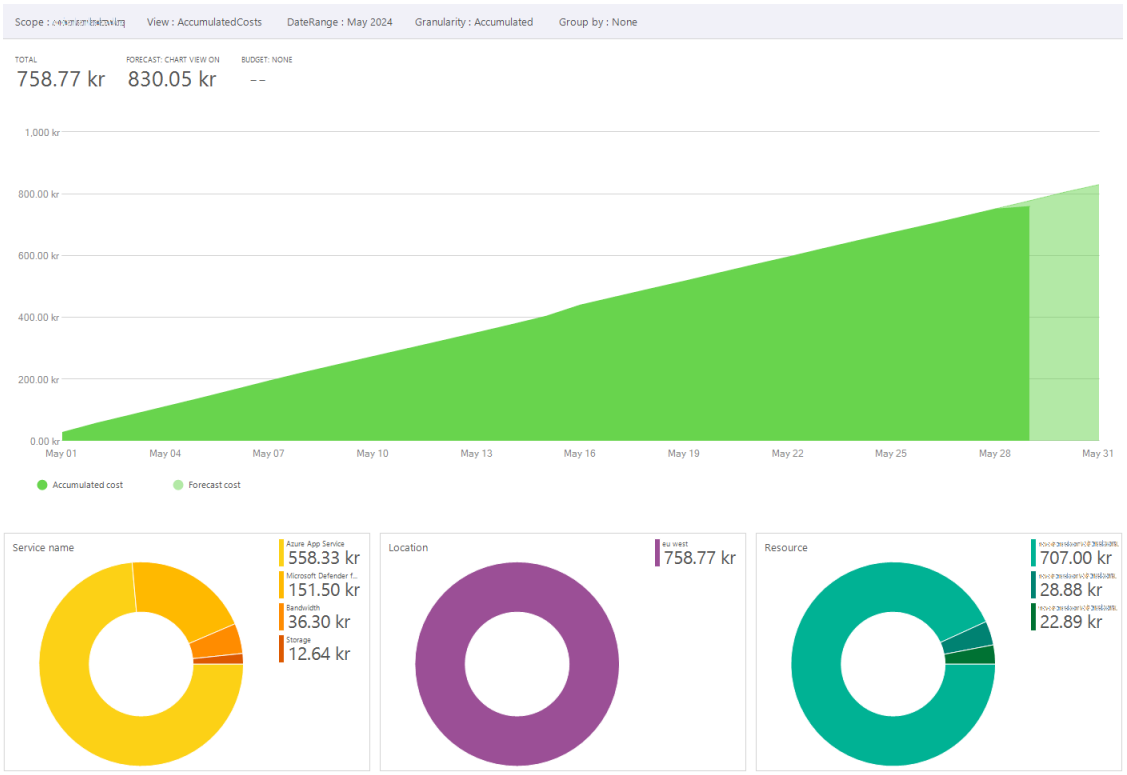

Figure 5.4: Monthly cost for genericsShop

Based on the forecast in Figure 5.2 trough 5.4 the average hourly cost can be calculated. To keep the analysis conservative, each month is assumed to have 30 days.

$$SEK\frac{1,392 + 1,341 + 830.05}{30\frac{d}{M} \times 24\frac{h}{d}} = SEK\frac{4.948}{h}$$

New Wave Group saves SEK 4.948 each hour the resources are not used. The cost effectiveness of the system can thus be calculated for each configuration using the formula below.

$$Net\ savings = -Monthly\ cost\ for\ the\ system + 4.948\ \times\ Hours\ of\ downtime$$

Since the net savings is dependent on the number of downtime hours for the resources, a graph is plotted displaying each configuration. See Figure 5.5.

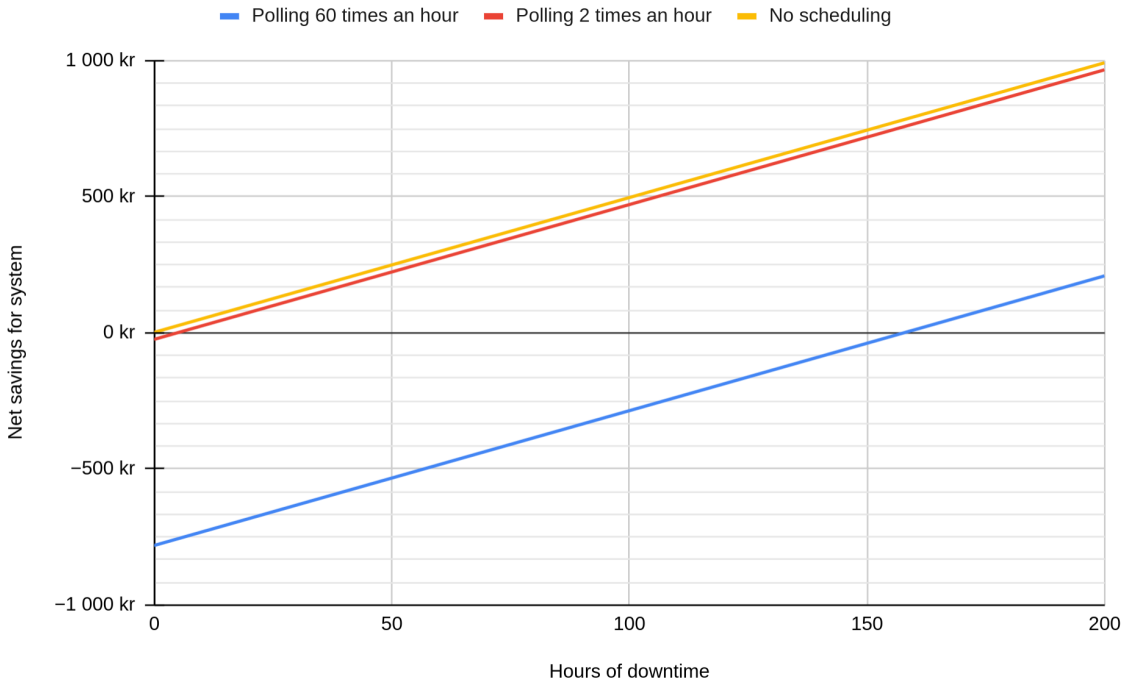

Figure 5.5: Net savings for system

The break-even for a responsive system is 156 downtime hours, after which the system becomes economically justifiable and results in savings. The realistic system configuration on the other hand only requires 5 downtime hours to break even, making it a comparatively cost effective option. The task-less system configuration does cost anything and is therefore also a cost effective solution, leading in savings from the start.

6
Discussion

This chapter provides a evaluation of the project, discussing the thought process behind the design choice and comparing alternative solutions. It also investigates potential future enhancement and encountered obstacles.

6.1 Design Choice, Bot vs Webhook

The decision to choose a webhook over a bot involved a series of difficult challenges driven by several factors. The first one was simplicity followed by efficiency and cost effectiveness. Both offer straightforward ways to integrate and interact with Azure but they differ in terms of operational aspects and cost aspects. During the course of the project, the importance of keeping costs down became clear.

The advantages of webhooks and Azure functions, apart from the integration simplicity, is immediate accumulation of commands which might not be possible in bots. Bots are more likely to have delayed processing and responses due to their complex nature. Considering simplicity, webhooks have a slight edge over bots. However, bots provide a more engaging and interactive experience, while webhooks only offer limited interactivity confined to predefined commands. A bot can be more intuitive, in the sense that it can assist and guide users through a process. A key aspect is that a bot would have the same core functionality as in the current solution. However, it would be considered a “dumb bot" due to its limited engagement and would also require significantly more overhead compared to a webhook.

For this project, a bot could be discarded not only due to its complexity but also because of its cost. A bot would need to be deployed and run continuously on Azure App Service in the cloud, with an estimated cost of at least SEK 3,000 monthly. The pricing is determined by the commissioned app service plan. Sending messages back and forth, however, is free. This situation mirrors the existing structure at the company, where the test environment is constantly up and running, and the task was to transition away from that approach. Rational reasoning would deem this improper but still a valid solution as it accomplishes all the set goals.

On the other hand, deploying a webhook-based system would avoid these costs. The pricing, as mentioned in section 4.8 and 5.1 is largely determined by the number of polls to the storage table. Customising polling intervals to match needs will provide best pricing. The execution of Azure function also has a set cost if the number of executions exceeds one million. In the current situation of the company, the chances of exceeding this number are very slim. The company rarely uses the test so the free tier subscription is more than sufficient.

When comparing the costs between using a bot service and using Azure Functions to implement the functionality, the bot service was found to be significantly more expensive than Azure Functions. After carefully weighing all aspects, both alternatives are great but the choice was ultimately influenced and determined by the costs.

6.2 Future development

One significant area of improvement that was omitted due to lack of time is the integration of database resource groups into the system. By simply adding the configurations in the JSON file the system can easily manage database resources as well. It is presumed that the database resource group is structured identically otherwise it would require customization. This addition would provide a more comprehensive control over the company's Azure ecosystem. The database resource is one of the most expensive resource groups and would significantly reduce costs if implemented. Even though integrating such a resource group was straightforward in this system the deciding factor rested with the other subgroup, as it would require a lot more overhead. Consequently the decision was made to eliminate it. Another reason for this decision was the GDPR since the database would grant us access to sensitive user information. Finding a secure way to process such information would require a considerable amount of time.

Another potential improvement to reduce costs is the implementation of durable functions. These functions operate similarly to the previously described Azure functions, but with the major difference being the saving of states between each function execution. As a result it would eliminate the need of a `table storage`, as task for scheduling could be saved as objects in a list. Lastly, the project's main goal was cutting on expenses. The system currently has support for instantiating three resource groups, which is just the tip of the iceberg. The figure below provides an overview of the projected monthly costs for New Wave Group's Azure subscriptions. Expanding support for more or even all resource groups would lower the cost significantly.

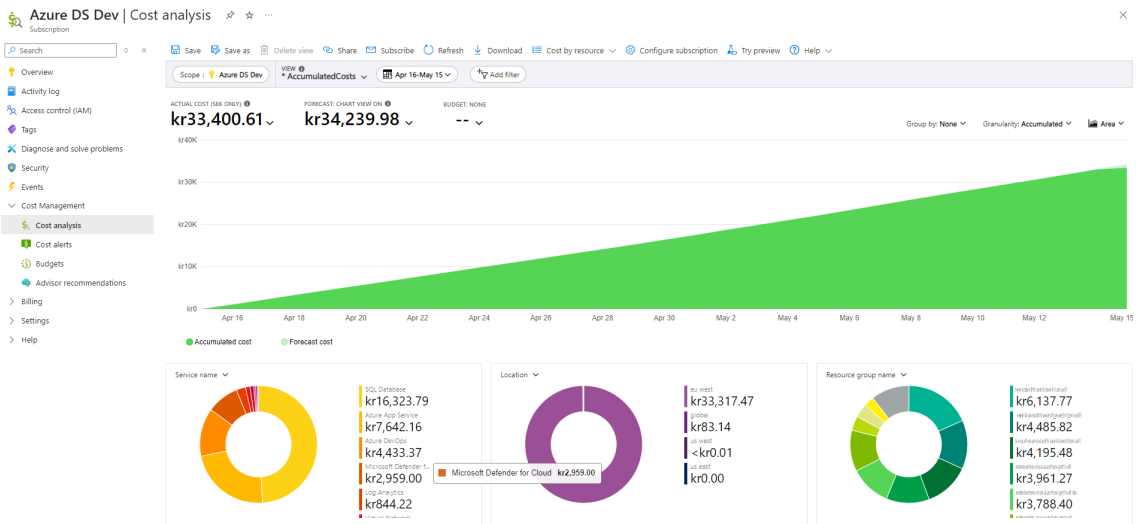

Figure 6.1: Cost analysis of resources within Azure for the development environment

6.3 Obstacles

The project throughout its course was not without challenges and difficulties. Throughout the development and implementation, obstacles were faced that required problem solving skills and fine tuning on our initial plans.

One major obstacle that was encountered during the project was delays due to clearances issues. Given New Wave's security protocols, gaining access to various parts in Azure took longer than expected. Considering it was New Wave Group's first time hosting a degree project thesis, they handled the situation with commendable professionalism. We managed to resolve issues in a timely manner through dialogue and regular meetings. Despite the smooth management, the impact of these challenges was still noticeable as it slightly slowed our progress. These delays were understandable and justifiable, given the need to protect the company's data and infrastructure. Another noteworthy obstacle during the project was the learning curve of these technologies. Since the project relied profoundly on understanding and learning C#, the Microsoft suite, including the Azure environment and Microsoft Teams, the burden was at times difficult to manage at once.

6.4 Ecological and Social Aspects

As stated in section 1.3 Objectives, one of the goals for the project is to be efficient. This means that calculations, processes, and data management should be conducted in a manner that does not unnecessarily burden the hardware. The primary reason is a cost issue, as a higher utilisation rate and loading of the servers entail increased marginal costs. However, an efficient program leads to lower energy consumption and, thereby, less energy used. It is important to point out, though, that the total energy consumption will probably increase as a result of the report. Since the test servers are not under load, they do not consume much energy, but the idea with this project is to increase the utilisation rate, which leads to increased emissions.

This project does not encompass any societal aspects. Given its scope, the outcomes are expected to impact only the stakeholders, indicating no direct effects on society at large.

6.5 Ethical Aspects

Technology today is advancing quickly which imposes certain requirements on security. The customer who purchases a service should feel extremely confident that their data is not exposed. With this in mind, it is a necessity that testing is emphasised properly. The advantages of testing on a large scale significantly minimise the risks. In addition, the code base becomes more secure, and the risk of potential intrusions or incorrect execution of code is notably reduced. Currently, there are no significant ethical aspects, apart from the security concerns, associated with the work. One could argue from a variety of normative schools of thought that the work plays an important role and is ethically founded. Let us take utilitarianism as an example: the work will increase the overall utility in society, developers can work on what they want, the product becomes safer and customers' data becomes more secure, it could lead to increased growth for the company which in turn benefits shareholders, even though emissions increase. After much discussion, however, the authors consider this to be far-fetched.

7
Conclusion

The purpose of the project was to create a cost effective system that improves the overall cloud resource management for New Wave Group. To achieve this, a CLI was created that lets users input commands in a Microsoft Teams channel. Each input is sent to an Azure function that is triggered using an outgoing webhook. The function processes the user command. A user command can trigger or schedule a pipeline run or provide help. The automation is handled by an Azure function which periodically checks scheduled pipeline runs, enabling a user to periodically start and delete a resource.

The design choices were mainly influenced by the focus on cost effectiveness. In this project a custom solution was created for a fraction of the cost for commercial alternatives. This custom solutions makes it easy to integrate with existing workflows, making sure it meets specific needs of the company. Using tools such as Azure functions has drawbacks. Azure functions are stateless and do not save any data between runs, making the implementation of smarter features cumbersome. In order to keep the costs down, features that require computationally intensive algorithms need to be disregarded. Creating a smart system is therefore challenging.

To improve the implementation, New Wave Group can build upon the current solution and create durable functions, removing the need for a database. POST requests need to contain all defined parameters to successfully trigger a pipeline run. To avoid this, it is advised to create a standard for the pipeline structure to avoid unnecessary rewrites of the Azure functions as new resources are added.

Bibliography

[1] Microsoft, "Azure functions http trigger." https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=python-v2%2Cisolated-process%2Cnodejs-v4%2Cfunctionsv2&pivots=programming-language-csharp. Accessed: 2024-04-15.

[2] Microsoft, "Timer trigger for Azure Functions." https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-timer?tabs=python-v2%2Cisolated-process%2Cnodejs-v4&pivots=programming-language-csharp. Accessed: 2024-04-15.

[3] N. W. Group, "Om New Wave Group." https://www.nwg.se/om-new-wave-group. Accessed: 2024-04-03.

[4] N. W. Group, "Våra varumärken." https://www.nwg.se/vara-varumarken/. Accessed: 2024-04-03.

[5] N. W. Group, "Synergier." https://www.nwg.se/om-new-wave-group/synergier/. Accessed: 2024-04-03.

[6] B. Pando and A. Dávila, "Software testing in the devops context: A systematic mapping study," *Programming and Computer Software*, vol. 48, p. 658–684, Dec 2022.

[7] T. Rangnau, R. v. Buijtenen, F. Fransen, and F. Turkmen, "Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines," *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, Oct 2020.

[8] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery," *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, May 2015.

[9] Ericsson, "CI/CD: IT meets networks." https://www.ericsson.com/48f6b1/assets/local/ci-cd/doc/cicd_it-meets-networks_2020.pdf. Accessed: 2024-04-08.

[10] GitHub, "About continuous integration." https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration. Accessed: 2024-04-08.

[11] Mozilla, "HTTP." https://developer.mozilla.org/en-US/docs/Web/HTTP. Accessed: 2024-04-15.

[12] E. Rescorla, "HTTP Over TLS." https://datatracker.ietf.org/doc/html/rfc2818. Accessed: 2024-04-15.

[13] Mozilla, "An overview of HTTP." https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Accessed: 2024-04-15.

[14] Mozilla, "HTTP headers." https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. Accessed: 2024-04-15.

[15] Mozilla, "HTTP Messages." https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Accessed: 2024-04-15.

[16] Mozilla, "HTTP request methods." https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Accessed: 2024-04-15.

[17] Mozilla, "GET." https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET. Accessed: 2024-04-15.

[18] Mozilla, "POST." https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST. Accessed: 2024-04-15.

[19] Mozilla, "DELETE." https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE. Accessed: 2024-04-15.

[20] R. Hat, "What is a webhook?." https://www.redhat.com/en/topics/automation/what-is-a-webhook. Accessed: 2024-04-15.

[21] GitHub, "About webhooks." https://docs.github.com/en/webhooks/about-webhooks. Accessed: 2024-04-15.

[22] Microsoft, "Create Outgoing Webhooks." https://learn.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/how-to/add-outgoing-webhook?tabs=urljsonpayload%2Cdotnet. Accessed: 2024-04-03.

[23] Microsoft, "Create Incoming Webhooks." https://learn.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/how-to/add-incoming-webhook?tabs=newteams%2Cdotnet. Accessed: 2024-04-15.

[24] ngrok, "What is ngrok?." https://ngrok.com/docs/what-is-ngrok/. Accessed: 2024-05-28.

[25] ngrok, "Ingress for dev/test environments." https://ngrok.com/use-cases/ingress-for-dev-test-environments. Accessed: 2024-05-28.

[26] requestly, "What is Ngrok and how does it work?." https://requestly.com/blog/what-is-ngrok-and-how-does-it-work/#:~:text=Ngrok%20is%20a%20cross%2Dplatform,on%20the%20internet%20through%20Tunnelling. Accessed: 2024-05-28.

[27] Microsoft, "Appar och tjänster." https://www.microsoft.com/sv-se/microsoft-365/products-apps-services. Accessed: 2024-04-08.

[28] Microsoft, "Ecosystem." https://opensource.microsoft.com/ecosystem/. Accessed: 2024-04-08.

[29] Microsoft, "A tour of the C# language." https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. Accessed: 2024-04-03.

[30] Microsoft, "What is Microsoft Teams?." https://support.microsoft.com/en-us/topic/what-is-microsoft-teams-3de4d369-0167-8def-b93b-0eb5286d7a29. Accessed: 2024-04-03.

[31] Microsoft, "Considerations for Teams integration." https://learn.microsoft.com/en-us/microsoftteams/platform/samples/integrating-web-apps. Accessed: 2024-04-03.

[32] Microsoft, "Vad är ett team i Microsoft Teams?." https://support.microsoft.com/sv-se/office/vad-%C3%

A4r-ett-team-i-microsoft-teams-f0a0f260-c494-4d54-bc3d-d2ce7a183a6e. Accessed: 2024-04-03.

[33] Microsoft, "Build webhooks and connectors." https://learn.microsoft.com/en-us/microsoftteams/platform/webhooks-and-connectors/what-are-webhooks-and-connectors. Accessed: 2024-04-03.

[34] Microsoft, "Create and add an outgoing webhook in Microsoft Teams." https://support.microsoft.com/en-us/office/create-and-add-an-outgoing-webhook-in-microsoft-teams-8e1a1648-982f-4511-b342- Accessed: 2024-04-03.

[35] TechTarget, "Azure DevOps." https://www.techtarget.com/searchwindowsserver/definition/Azure-DevOps-formerly-Visual-Studio-Team-Services. Accessed: 2024-04-08.

[36] Microsoft, "What is Azure DevOps?." https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?toc=%2Fazure%2Fdevops%2Fget-started%2Ftoc.json&view=azure-devops. Accessed: 2024-04-08.

[37] Microsoft, "Overview of services." https://learn.microsoft.com/en-us/azure/devops/user-guide/services?view=azure-devops. Accessed: 2024-04-08.

[38] Microsoft, "Azure Pipelines." https://azure.microsoft.com/en-us/products/devops/pipelines#overview. Accessed: 2024-04-05.

[39] T. Y. Project, " Accessed: 2024-05-25.

[40] Microsoft, "Yaml schema reference for azure pipelines." https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines&viewFallbackFrom=azure-%20pipelines. Accessed: 2024-05-25.

[41] Codefresh, "Ci/cd process: Flow, stages, and critical best practices." https://codefresh.io/learn/ci-cd-pipelines/ci-cd-process-flow-stages-and-critical-best-practices/#:~:text=The%20CI%2FCD%20pipeline%20combines,build%2C%20test%2C%20and%20deploy. Accessed: 2024-04-05.

[42] Microsoft, "Design the pipeline." https://learn.microsoft.com/en-us/training/modules/create-multi-stage-pipeline/2-design-the-pipeline. Accessed: 2024-04-05.

[43] Microsoft, "Define resources in YAML." https://learn.microsoft.com/en-us/azure/devops/pipelines/process/resources?view=azure-devops&tabs=schema. Accessed: 2024-04-05.

[44] Microsoft, "Plan a release pipeline by using Azure Pipelines." https://learn.microsoft.com/en-us/training/modules/create-release-pipeline/3-plan-release-pipeline. Accessed: 2024-04-05.

[45] Microsoft, "Deploy to App Service using Azure Pipelines." https://learn.microsoft.com/en-us/azure/app-service/deploy-azure-pipelines?tabs=yaml. Accessed: 2024-04-05.

[46] Microsoft, "Task types & usage." https://learn.microsoft.com/en-us/azure/devops/pipelines/process/tasks?view=azure-devops&tabs=yaml. Accessed: 2024-04-05.

[47] Microsoft, "Azure Functions overview." https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp. Accessed: 2024-04-03.

[48] Microsoft, "Differences between the isolated worker model and the in-process model for .NET on Azure Functions." https://learn.microsoft.com/en-us/azure/azure-functions/dotnet-isolated-in-process-differences. Accessed: 2024-04-15.

[49] Microsoft, "Guide for running C# Azure Functions in the isolated worker model." https://learn.microsoft.com/en-us/azure/azure-functions/dotnet-isolated-process-guide?tabs=windows. Accessed: 2024-04-15.

[50] Microsoft, "Develop C# class library functions using Azure Functions." https://learn.microsoft.com/en-us/azure/azure-functions/functions-dotnet-class-library?tabs=v4%2Ccmd. Accessed: 2024-04-15.

[51] Microsoft, "Azure Functions pricing." https://azure.microsoft.com/en-us/pricing/details/functions/. Accessed: 2024-04-03.

[52] Microsoft, "Azure Functions triggers and bindings concepts." https://learn.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=isolated-process%2Cpython-v2&pivots=programming-language-csharp. Accessed: 2024-04-03.

[53] Microsoft, "Pricing calculator." https://azure.microsoft.com/en-us/pricing/calculator/. Accessed: 2024-05-25.

[54] Microsoft, "Subscriptions, licenses, accounts, and tenants for microsoft's cloud offerings." https://learn.microsoft.com/en-us/microsoft-365/enterprise/subscriptions-licenses-accounts-and-tenants-for-microsoft-cloud-offerings?view=o365-worldwide. Accessed: 2024-05-25.

[55] Microsoft, "Pay as you go with azure—plus get free services." https://azure.microsoft.com/en-us/pricing/purchase-options/pay-as-you-go/. Accessed: 2024-05-25.

[56] Microsoft, "What is azure table storage?." https://learn.microsoft.com/en-us/azure/storage/tables/table-storage-overview. Accessed: 2024-05-25.

[57] Microsoft, "Table storage." https://azure.microsoft.com/en-us/products/storage/tables. Accessed: 2024-05-25.

[58] Microsoft, "Understanding the Table service data model." https://learn.microsoft.com/en-us/rest/api/storageservices/understanding-the-table-service-data-model. Accessed: 2024-05-29.

[59] Microsoft, "bool (C# reference)." https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/bool. Accessed: 2024-05-29.

[60] Microsoft, "DateTime Struct." https://learn.microsoft.com/en-us/dotnet/api/system.datetime?view=net-8.0&redirectedfrom=MSDN. Accessed: 2024-05-29.

[61] Microsoft, "char (C# reference)." https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/char. Accessed: 2024-05-29.
[62] Microsoft, "Azure Tables bindings for Azure Functions." https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-table?tabs=isolated-process%2Ctable-api%2Cextensionv3&pivots=programming-language-csharp. Accessed: 2024-04-15.
[63] Microsoft, "Table storage pricing." https://azure.microsoft.com/en-us/pricing/details/storage/tables/. Accessed: 2024-05-25.
[64] Microsoft, "Redundans i azure storage." https://learn.microsoft.com/sv-se/azure/storage/common/storage-redundancy. Accessed: 2024-05-25.
[65] Microsoft, "RESTful web API design." https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design. Accessed: 2024-04-03.
[66] IBM, "What is a REST API?." https://www.ibm.com/topics/rest-apis. Accessed: 2024-04-03.
[67] Microsoft, "Azure REST API reference." https://learn.microsoft.com/en-us/rest/api/azure/#how-to-call-azure-rest-apis-with-curl. Accessed: 2024-04-03.
[68] Microsoft, "Pipelines - List." https://learn.microsoft.com/en-us/rest/api/azure/devops/pipelines/pipelines/list?view=azure-devops-rest-7.2. Accessed: 2024-04-03.
[69] Microsoft, "Pipelines - Get." https://learn.microsoft.com/en-us/rest/api/azure/devops/pipelines/pipelines/get?view=azure-devops-rest-7.2. Accessed: 2024-04-03.
[70] Microsoft, "Pipelines - Create." https://learn.microsoft.com/en-us/rest/api/azure/devops/pipelines/pipelines/create?view=azure-devops-rest-7.2. Accessed: 2024-04-03.
[71] P. Flewelling, *The Agile Developer's handbook*. Packt Publishing, 2018.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY