



**Università
di Genova**

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

DevOptimizeAI: a GPT based tool for supporting developers in a DevOps scenario

Alessandro Penco

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
<https://www.dibris.unige.it/>



**Università
di Genova**

MSc Computer Science
Data Science and Engineering Curriculum

DevOptimizeAI: a GPT based tool for supporting developers in a DevOps scenario

Alessandro Penco

Advisor: Prof. Daniele D'Agostino, Prof. Gianna Reggio
Examiner: Prof. Matteo Dell'Amico

December, 2023

Abstract

The main contribution of this thesis is to address the pressing challenge of maintaining high-quality software in large teams with developers of different experience levels. In particular, it presents an evaluation of the effectiveness of GPT models, coupled with Apache JMeter, in supporting developers in a DevOps scenario of REST API development via Azure Functions. Furthermore, the thesis aims to identify the potential benefits that can be gained from using the discussed approach in the broader software development scenario, e.g. to identify areas in need of improvement or resolve logical errors in code and guide developers through the code implementation process.

To address this challenge, this thesis discusses the development of a tool, called DevOptimizeAI, that can enhance CI pipelines in the previously described scenario. The tool will consist of two basic parts: a web app form for creating YAML code to be copy&pasted to the developer's pipeline and a pair of pipeline stages. The former runs tests with JMeter and prepares the results, the latter analyzes the results properly adapted and merged to the Azure Function code via a GPT model and delivers code improvement recommendations to the developer.

Results show that the tool has the potential to significantly improve the efficiency and quality of Azure Functions by always providing tailored recommendations as well as speeding up development time, which will ultimately lead to better products and services for end users.

Table of Contents

Chapter 1	Introduction and motivation	7
Chapter 2	Background and related works	9
2.1	DevOps	9
2.2	Generative AI	16
2.3	Performance testing Tools	22
2.4	Related Works	27
Chapter 3	DevOptimizeAI Requirements and Design	34
3.1	DevOptimizeAI mission	34
3.2	Requirements	36
3.3	Design	42
Chapter 4	DevOptimizeAI Implementation	51
4.1	Configuration Form	51
4.2	DevOptimizeAI pipeline stages	55
4.3	GPT response formatting	60
Chapter 5	Experimental Results	64
5.1	Azure Functions Test Code Implementation	66
5.2	Acceptance tests	73
5.3	Validation tests	83

Chapter 6	Conclusions and future works	102
6.1	Conclusions	102
6.2	Future works	104
List of Tables		105
List of Figures		106
Bibliography		108
Appendix A	Template YAML code	I
A.1	JMeter YAML code	I
A.2	OpenAi YAML code	III
Appendix B	C# function code	VI
B.1	GetMean code	VI
B.2	GetAPI code	IX
B.3	PostAPI code	XI
B.4	CalculatePrimeNumbers code	XIII
Appendix C	JMeter Test Plans	XV
C.1	GetMean Test plan	XV
C.2	GetAPI Test plan	XX
C.3	PostAPI Test plan	XXVI
C.4	CalculatePrimeNumbers Test plan	XXXII
Appendix D	Prompt Engineering component	XXXIX
D.1	JMeter result files	XXXIX
D.2	Generated JMeter input for GPT model	XLI
D.3	Input for GPT model	XLII

Appendix E	Test with different program without prompt engineering	XLVII
E.1	Test results with GPT-3	XLVII
E.2	Program test results with GPT4	XLIX
Appendix F	Prompt Test	LII
F.1	Prompt Test using only system message	LII
F.2	Prompt Test using only user message	LV
F.3	Prompt Test using both user and system message	LXIII
Appendix G	CPU-Intensive Test	LXX
G.1	CalculatePrimeNumbers Test	LXX
Appendix H	Behavioural Test	LXXVI
H.1	Prompt Test providing prompt to different developer types	LXXVI
H.2	Prompt Test providing hint for the model	LXXX
H.3	Model behaviour with previous successful messages example	LXXXII
H.4	Prompt Test using feedback provided from the model for GetMean API . .	LXXXIV
Appendix I	ChatGPT Response	XCII

Chapter 1

Introduction and motivation

In the current digital era, web applications and APIs form the backbone of many businesses and organizations worldwide. These digital tools are not just interfaces; they are vital instruments that drive customer engagement, streamline business operations, and contribute to the creation of value. Within this context, the performance of web applications and APIs, particularly those developed through Azure Functions in a DevOps environment, is vital for ensuring user satisfaction, operational efficiency, and overall business success.

However, maintaining high-quality, efficient, and reliable software in such a dynamic environment presents significant challenges. Large development teams, often consisting of developers with varying degrees of experience, can struggle to maintain consistency in code quality and efficiency. Moreover, the pressure of swift development and deployment cycles inherent in DevOps can exacerbate these issues, leaving little room for comprehensive code optimization and performance tuning.

Recognizing these challenges, this Master's thesis introduces DevOptimizeAI, an innovative tool designed to support developers in enhancing the efficiency and quality of Azure Functions. DevOptimizeAI brings together Generative Pretrained Transformer (GPT) models and Apache JMeter, offering an automated solution for code optimization in a DevOps setting. The tool serves to identify areas in need of improvement and guides developers through the code implementation process, thereby enhancing overall software quality.

DevOptimizeAI integrates seamlessly into the Continuous Integration (CI) pipeline and comprises two core components. The first is a form for creating YAML code, which developers can easily incorporate into their existing pipeline. The second component consists of a pair of pipeline stages. The first stage conducts performance tests with JMeter and prepares the results. The second stage leverages a GPT model to analyze the test results and the Azure Function code, providing personalized code improvement recommendations

to the developer.

The primary objective of this thesis is to evaluate the tangible benefits of leveraging GPT models and Apache JMeter as a support system in a DevOps scenario of REST API development via Azure Functions.

The thesis aims to show the potential benefits of this approach in broader software development scenarios, enhancing overall software quality, and leading to more efficient, reliable, and robust web applications and APIs.

The research also explores the practical implementation of DevOptimizeAI in a CI pipeline, shedding light on how such a tool can streamline and automate the code optimization process.

Ultimately, this thesis contributes to the ongoing discourse on software development best practices, demonstrating how innovative solutions like DevOptimizeAI can address pressing challenges faced by development teams, particularly in the context of maintaining high-quality software in large teams with varying experience levels, and lead to better products.

The thesis is organized as follows.

Chapter 2 provides background information and relevant related works about DevOps and Azure Functions, generative AI and ChatGPT and then benchmarking tools and Apache JMeter.

In Chapter 3 outlines the specific requirements and design considerations for developing DevOptimizeAI, detailing its mission and expected functionalities.

Chapter 4 discuss about the practical implementation of each of the previously designed DevOptimizeAI components going into details.

Chapter 5 focuses on experimenting DevOptimizeAI. Initially it describe the major function with the tool will be tested. Then will be done different test: from testing different function with different tool configuration, to evaluating DevOptimizeAI behaviour in difference circumstances, passing through tests on functions with different objectives and complexity.

Finally, Chapter 6 draws conclusions and proposes possible implications and future work.

Chapter 2

Background and related works

This chapter will initially present DevOps and CI/CD explaining what they are and the advantages of using them. It also present the Microsoft Azure service and other important concept and services that will be used in the thesis.

It discuss the AI topics of this thesis such as generative AI, Large Language Models (LLMs), and OpenAI models. It make a dive in the ChatGPT tool and explain some of its important features and also show some related work about AI and DevOps.

Finally it will deal with the topic of performance testing tools in the field of computer science, going to compare some in the field of web applications, focusing on JMeter. The last part discuss on the situation about the performance testing in relation to AI.

2.1 DevOps

In the fast-paced arena of modern software development, the convergence of DevOps and Continuous Integration and Continuous Delivery (CI/CD) has emerged as a pivotal paradigm. DevOps removes barriers between development and operations, promoting collaboration and agility. Meanwhile, CI/CD automates the software release process, enabling frequent, reliable, and efficient deployment. This section outlines the fundamentals of DevOps and CI/CD, their principles, and their transformative impact on software development methodologies.

What is DevOps?

There is no globally agreed definition to describe what *DevOps* is.

For example, Microsoft provides the following definition: "DevOps combines development

(Dev) and operations (Ops) to unite people, process, and technology in application planning, development, delivery, and operations”[25]. Instead Amazon Web Services (AWS) provides its definition, which is semantically equal, but syntactically different; and is: ”DevOps is the combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes”[13]. Or again, ChatGPT provides this version: ”DevOps is a set of practices and principles that aims to improve the software development process by increasing collaboration, automation, and feedback between software developers and IT operations teams”.

Although they have different syntaxes, they are not different versions or applications of DevOps and all have the same purpose: The goal of DevOps is to shorten the software development lifecycle, from the initial idea to production deployment, by making the process more efficient and effective. DevOps brings together development and operations teams

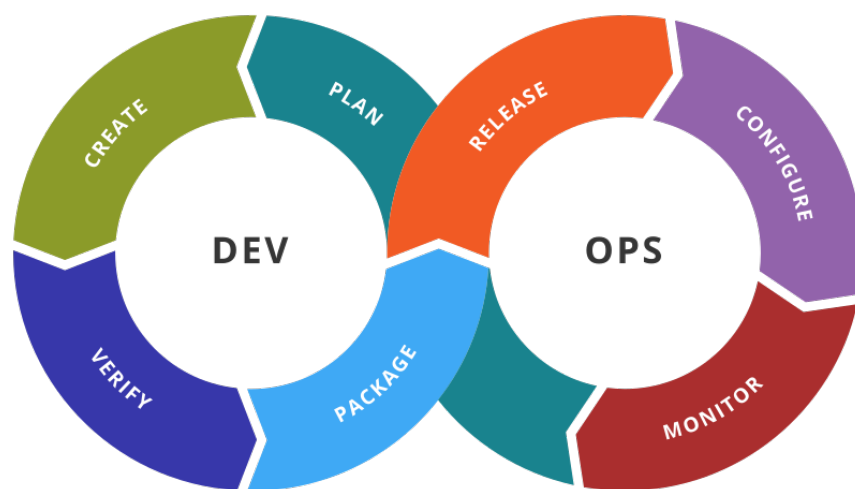


Figure 2.1: DevOps

to work together throughout the entire software development process as represented in Figure 2.1. This collaboration helps to eliminate silos and ensures that everyone is working towards the same goals. DevOps also emphasizes automation, with the use of tools and processes to automate repetitive tasks and increase the speed and accuracy of the software development process.

Another key aspect of DevOps is the use of continuous feedback and monitoring. By constantly monitoring and analyzing software performance, teams can identify issues and make improvements quickly. This helps to ensure that software is reliable, secure, and meets the needs of end-users.

In a CI/CD scenario DevOps is implemented through a pipeline that automates the process

of continuous integration and continuous delivery (CI/CD).

DevOps pipeline

A *DevOps pipeline* is a set of automated processes that facilitate the integration, testing, deployment, and delivery of software applications. It is a key component of a DevOps approach to software development, enabling organizations to automate the software delivery process and ensure that software is delivered rapidly and reliably.

In this context, multiple developers are collaborating to produce different software components that need to be integrated and delivered quickly and efficiently. The DevOps pipeline begins with the code being pushed to a version control system, such as Git, which triggers the continuous integration stage.

In the continuous integration stage, the code is compiled, built, and tested using automation tools such as Azure Pipelines, Jenkins, Travis CI, or CircleCI. If the tests pass, the code moves to the continuous delivery stage, where it is deployed to a staging environment for further testing and validation.

Continuous integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control[22].

CI encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion. Committing code triggers an automated build system to grab the latest code from the shared repository and to build, test, and validate the full main branch (also known as the trunk).

A *pipeline* defines the continuous integration process for the application. It's made up of steps called tasks. You can think of it as a script that defines how your build, test, and deployment steps are run.

The pipeline runs when you submit code changes. You can configure the pipeline to run automatically, or you can run it manually. You connect your pipeline to a source repository like GitHub, Bitbucket, or Subversion. One of our tasks for this sprint is to start using GitHub. So we'll use GitHub for this project.

A build agent builds or deploys the code. When your build or deployment runs, the system begins one or more jobs. An agent is an installable software that runs one build or deployment job at a time.

The final product of the pipeline is a build artifact. Think of an artifact as the smallest compiled unit that we need to test or deploy the app. For example, an artifact can be:

- A Java or .NET app packaged into a .jar or .zip file.
- A C++ or JavaScript library.

- A virtual machine, cloud, or Docker image.

Once the code has been validated in the staging environment, it moves to the final stage of the pipeline: continuous deployment. In this stage, the code is automatically deployed to the production environment using tools such as Kubernetes or Docker Swarm.

Throughout the pipeline, automated testing and monitoring tools are used to ensure that the code meets the required quality standards and is functioning correctly in production. This helps to detect and fix issues quickly, ensuring that the software is delivered reliably and efficiently.

By using a DevOps pipeline, organizations can reduce the time and effort required to deliver high-quality software while also improving the overall reliability and security of their software applications.

Microsoft Azure

Microsoft Azure, commonly known as Azure, is a cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers. It provides a broad range of services including those for computing, analytics, storage, and networking. Users can choose and configure these services to meet their specific needs [36].

Azure Subscription

An Azure subscription is a logical unit of Azure services that is linked to an Azure account. In essence, an Azure subscription is an agreement with Microsoft to consume and pay for cloud resources. Each Azure subscription can have a separate billing setup, so you can have different subscriptions on your account for different departments, projects, or billing requirements [12].

Subscriptions help you organize access to cloud service resources, providing a way to manage the resources that your applications use. They allow you to control and manage costs, monitor usage, and report on your organization's cloud resource use.

In the context of Azure DevOps, you can use Azure subscriptions to manage and deploy your resources. For example, you can create an Azure Resource Manager service connection with your Azure subscription in Azure DevOps, allowing your Azure DevOps pipelines to authenticate and deploy resources to Azure.

An Azure subscription is also required for using Azure Key Vault, a service that safeguards cryptographic keys and secrets used by cloud applications and services. Once you have

an Azure subscription, you can create a Key Vault in the Azure portal and begin adding secrets to it, which can then be accessed securely by your applications.

Azure DevOps

In the context of DevOps, Azure offers a wide array of services to support the DevOps approach. Azure DevOps, a service offered by Microsoft, provides developer services to support teams to plan work, collaborate on code development, and build and deploy applications. Azure DevOps supports a range of applications types and is extensible and fully featured to support enterprises [42].

Azure Pipelines

Azure DevOps includes *Azure Pipelines* is a cloud service you can use to automatically build, test, and deploy your code project. You can also make it available to other users, and it works with just about any language or project type [23].

Furthermore, Microsoft Azure includes a vast range of services that can be used in a DevOps pipeline. From Azure Repos for Git-based source control, Azure Boards for work tracking, Azure Test Plans for testing applications, to Azure Artifacts for hosting package feeds, Azure provides all the tools necessary for implementing a robust and efficient DevOps pipeline.

Service Endpoint Connection

Service Endpoint Connection in Azure DevOps is a secure way of connecting to and accessing resources inside an Azure environment. It encapsulates the details of a connection to a particular service, providing a secure, reusable connection to external systems or services. This is crucial in a DevOps context, where you might need to connect to different services like Azure Resource Manager, Docker Registry, or others. Service endpoints can be used in Azure Pipelines to connect to any of these services securely and execute tasks during a pipeline run [27].

In the context of this thesis, the Azure DevOps pipeline will interact with a GitHub repository to access the Azure Functions code and with Azure Key Vault to manage secrets needed to call a GPT model.

Let's consider a scenario where your Azure DevOps pipeline needs to interact with a GitHub repository for source code.

1. In Azure DevOps, navigate to Project Settings, then select Service connections under Pipelines.
2. Click on New service connection, then select GitHub from the drop-down list.

3. If prompted, authenticate with your GitHub account credentials.
4. After successful authentication, you can select the specific repository you want to connect with.
5. Click on Save.

Now, this GitHub service endpoint can be referenced in your Azure Pipelines configuration. This allows Azure Pipelines to securely interact with your GitHub repository, pulling the necessary code for builds and deployments.

Azure Key Vault

Azure Key Vault is a cloud-based service provided by Microsoft's Azure platform. It is designed to safeguard cryptographic keys and secrets (like tokens, passwords, certificates, API keys) used by your cloud applications and services[7].

Azure Key Vault provides a way to securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets. It is backed by hardware security modules (HSMs), providing a secure and scalable way to manage and control access to these secrets.

Instead of having application and service code that contains sensitive information, Azure Key Vault allows applications to use these secrets without exposing them. Secrets and keys can be created and controlled through Azure Key Vault, and then accessed through a secured HTTP API when needed.

Additionally, Azure Key Vault allows users to manage and enforce access policies, and to track usage of secrets using logging mechanisms. This can be useful for auditing and compliance. Azure Key Vault can also handle tasks such as secret rotation and expiration.

Using Azure Key Vault

To use Azure Key Vault, you create a vault, and store any secrets that your application uses in that vault. Applications or users must authenticate with Azure Active Directory (AD) before they can access the secrets in Azure Key Vault, providing an additional layer of security.

Once authenticated, applications can access a secret (for example, a connection string or an API key) by using a URI. This means the secret itself is not included in the application code, and the secret is delivered securely to the application.

To connect Azure KeyVault with Azure DevOps, follow these steps:

1. In Azure DevOps, navigate to Project Settings, then select Service connections under Pipelines.
2. Click on New service connection, then select Azure Resource Manager from the drop-down list.
3. Provide the necessary details such as the Subscription ID, Subscription Name, Service Principal Id, Service principal key, and Tenant ID. These details allow Azure Pipelines to authenticate and interact with Azure KeyVault.
4. Under 'Resource Group', select the Resource Group where your KeyVault is located. Under 'Azure Key Vault', select the specific KeyVault you want to access.
5. Click on Save.

Now, this Azure KeyVault service endpoint can be referenced in your Azure Pipelines configuration. This allows Azure Pipelines to securely fetch secrets from your Azure KeyVault during pipeline runs.

These examples illustrate how service endpoints in Azure DevOps can be used to securely interact with external services, like GitHub and Azure KeyVault. By encapsulating connection details, they provide a way to securely interact with these services without exposing sensitive information, enhancing the security and efficiency of your DevOps operations.

Azure Functions

Azure Functions is a serverless computing service provided by Microsoft as part of the Azure cloud platform. It allows developers to write and execute pieces of code in the cloud without having to explicitly provision or manage infrastructure [14].

Serverless computing is an event-driven model, and with Azure Functions, these events could be anything from HTTP requests, timer-based triggers, to changes in a database or a message queue. The function is the unit of deployment and scaling, and it contains the code to be run in response to the event trigger.

Azure Functions operate on a "write and run" model. You write the code for the function in a language supported by Azure (such as C#, Java, JavaScript, Python, or PowerShell), and then Azure takes care of the rest. It executes the code when the specified event happens and then scales out to handle high loads when needed, and scales in when demand is low.

Each function has associated bindings. Bindings are ways of declaratively connecting other resources to the function. Input bindings are data that the function receives, and output bindings are data the function sends out. For example, a function could have an input binding to Azure Storage blob (which triggers the function when a blob is added) and an output binding to send a message to an Azure Queue Storage.

Azure Functions offer several benefits. They eliminate the need to manage servers, allowing developers to focus on writing code. They offer automatic scaling and you only pay for the time your code runs, making them cost-efficient. They support a variety of programming languages and offer seamless integration with other Azure services.

However, Azure Functions also have some weaknesses. Cold start can be an issue where a function hasn't been invoked for a period of time, and then it takes longer to run when it's triggered again. Debugging and testing Azure Functions, especially complex ones, can be challenging. Stateless functions (which most Azure Functions are) don't retain data from one execution to the next, which might not be ideal for some use cases.

2.2 Generative AI

Generative AI is a subfield of artificial intelligence that focuses on the development of machines and algorithms that can create or generate new content, such as images, music, or text, without explicit human input, but only providing to the algorithm some information about what it is wanted to be generate (prompt), more sophisticated and precise information are provided to the algorithm and it is more likely to obtain the desired output. Generative AI can be used to generate new content that is both realistic and diverse, allowing organizations to create unique and engaging content for their audiences. Generative AI is being used in a wide range of applications, including video game development, graphic design, and content creation [37].

One example of generative AI is the use of generative adversarial networks (GANs) to generate realistic images. GANs consist of two neural networks: a generator and a discriminator. The generator creates new images based on random noise input, while the discriminator tries to distinguish between real and generated images. Through an iterative process of training, the generator learns to create images that are increasingly difficult for the discriminator to distinguish from real images. This process can be used to generate realistic images of objects, people, and even entire scenes.

Generative AI is being increasingly used in the code development area to automate various aspects of software development, including code completion, code generation, and bug fixing. One of the main advantages of using generative AI in code development is that it can help reduce the time and effort required to write high-quality code [37].

For example, AI-based code completion tools can suggest code snippets and functions that are likely to be relevant based on the context of the code, enabling developers to write code more quickly and efficiently. Similarly, AI-based code generation tools can generate new code based on existing code or user input, making it easier to create new applications and software features. Furthermore, generative AI can be used to identify and fix bugs in code,

allowing developers to create more robust and reliable software applications.

Generative AI is a vast topic, this thesis limit the analysis and use of generative AI that is powered by deep learning techniques that rely on a type of AI model called a language model.

Large Language model (LLM)

“In very simple terms, the LLM is, at its core, just a document completion model. For training it was given partial documents and it learned how to complete them one token at a time. Therefore, the art of prompt crafting is really all about creating a ‘pseudo-document’ that will lead the model to a completion that benefits the customer,” John Berryman[41], a senior researcher of machine learning on the Model Improvements team explains.

A language model is a type of model that is specifically designed to understand and generate natural language. It is trained on large amounts of text data and learns the statistical patterns and relationships between words and phrases in a given language. The primary goal of a language model is to predict the likelihood of a particular word or phrase given the context of the language. The difference between language model and large language model is that an LLM is the evolution of the language model concept in AI that dramatically expands the data used for training and inference.

One key difference between language models and generative models is their purpose. Language models are specifically designed to understand and generate natural language, while generative models can be used to generate a wide range of data types. Another key difference is the training process. Language models are typically trained on large amounts of text data, while generative models can be trained on a wide range of data types, depending on the application. Additionally, the evaluation metrics for language models and generative models are different. For language models, the primary metric is often perplexity, which measures how well the model can predict the next word given the context. For generative models, the primary metric is often the quality of the generated data, which can be evaluated using a range of metrics specific to the data type being generated.

One of the most advanced and widely-used language models in the field of Generative AI is the GPT (Generative Pre-trained Transformer) series developed by OpenAI. GPT models are pre-trained on vast amounts of data and can be fine-tuned to generate high-quality text content, such as articles, stories, or even code.

GPT series and OpenAI

OpenAI is an AI research and deployment company, its mission is to ensure that artificial general intelligence benefits all of humanity [29]. One of their notable achievements is the creation of the GPT language models, which have been at the forefront of natural language processing research.

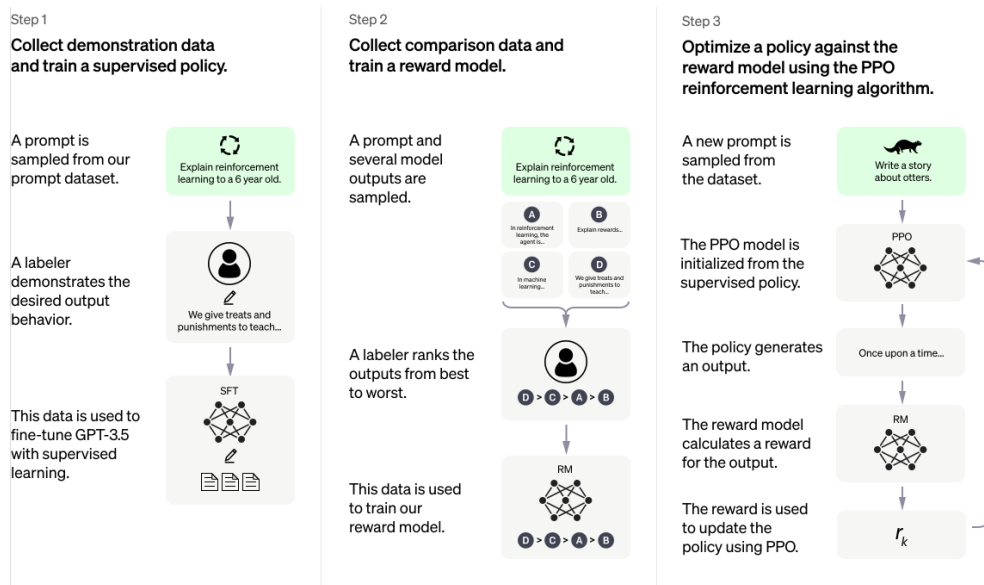


Figure 2.2: ChatGPT Methodology Diagram

ChatGPT is a sibling model to InstructGPT, which is trained to follow an instruction in a prompt and provide a detailed response. ChatGPT is fine-tuned from a model in the GPT-3.5 series, which finished training in early 2022 [30]. The methodology used to train this model is shown in Figure 2.2.

It is designed to generate human-like responses to natural language prompts, making it an ideal tool for a wide range of applications, including chatbots, virtual assistants, and customer support systems. ChatGPT is trained on a vast corpus of text data, enabling it to generate coherent and contextually appropriate responses to a wide range of queries. It is capable of understanding the nuances of human language and can generate responses that are indistinguishable from those of a human being. ChatGPT has been used in a variety of applications, including customer support, language translation, and content creation, and its versatility and effectiveness make it an important tool for businesses and organizations looking to automate their customer service and communication processes.

The ChatGPT API allows developers to integrate the model’s capabilities into their applications, enabling them to generate high-quality, natural-sounding text and have conversations with users in a variety of contexts.

ChatGPT

ChatGPT is designed as an application of GPT-3.5-turbo specifically for generating conversational responses. It’s mainly used in interactive scenarios, such as drafting emails, writing

Models	<i>Bard</i>	<i>ChatGPT</i>	<i>GPT-4</i>	<i>LLaMa</i>
Audience	Users interested in a natural language generating/processing chatbot that can answer human questions	Users interested in a conversational chatbot that can be used for various natural language processing tasks, such as question answering, conversation, and text generation	Users interested in a powerful large-scale unsupervised language model that can generate human-like text and complete a wide variety of tasks	AI developers interested in a powerful large language model
Offers API	No	Yes	Yes	No
Pricing	Free	Free	\$0.0200 per 1000 tokens	Free

Table 2.1: Most popular commercial LLM comparison

code or content, answering questions, creating conversational agents, tutoring, translating languages, simulating characters for video games, and more. For general users, interacting with ChatGPT typically involves a straightforward, user-friendly interface where users can provide their input and receive responses from the model [24].

GPT-3.5-turbo

GPT-3.5-turbo refers to the AI model that powers applications like ChatGPT. For developers or researchers who use the OpenAI API directly, they can utilize the power of GPT-3.5-turbo for more than just conversation. They can use it for various complex tasks such as text generation, summarization, translation, and more. To use GPT-3.5-turbo via the API, users need to construct API calls, which requires technical knowledge. Developers can also modify various parameters like temperature and max tokens to influence the model’s output.

So, in practical terms, the difference between using ChatGPT and GPT-3.5-turbo depends on the user’s technical knowledge and their specific needs. ChatGPT is more suited for general users looking for conversational AI, while GPT-3.5-turbo offers developers more flexibility and control over the AI’s capabilities [2].

Strengths and Limitations of ChatGPT

ChatGPT has several strengths. It can generate creative and diverse outputs, it can

understand and generate text in multiple languages, and it can be fine-tuned on specific tasks to provide more accurate responses [24].

However, it also has limitations. It can sometimes write incorrect or nonsensical answers. The model is sensitive to slight tweaks in the input phrasing. It also tends to be excessively verbose and overuses certain phrases. It doesn't always ask clarifying questions when faced with ambiguous queries, often guessing the user's intent [24].

In terms of safety, while significant efforts have been made to refuse inappropriate requests, it might sometimes respond to harmful instructions or exhibit biased behavior.

Prompt Engineering

Prompt engineering is a critical aspect of effectively using models like ChatGPT. It involves strategically crafting the input, or "prompt", to guide the AI model towards producing the desired output. It's not just about what you ask, but how you ask it. The phrasing, context, specificity, and structure of the prompt can all significantly influence the model's response.

The importance of effective prompt engineering cannot be overstated. With a well-crafted prompt, you can guide the AI model to generate more accurate, relevant, and useful responses. It can help in reducing the model's tendency to generate verbose or off-topic responses, and increase the likelihood of the model providing a direct and concise answer.

For instance, if a user is seeking a specific piece of information, a more structured and explicit prompt can help in getting a direct answer. Additionally, system-level instructions, such as "You are an assistant that speaks like Shakespeare", can be used to guide the model's tone and style in the response.

However, crafting effective prompts often requires a deep understanding of the model and its behavior, as well as trial and error. It could be challenging for users who are not familiar with the model's complexity. This underlines the need for more user-friendly interfaces and tools that can assist users in prompt crafting, as well as continued research to make models more robust to variations in prompt phrasing.

System and User Messages in ChatGPT and GPT Models via API

In the context of the GPT models, specifically within a conversation, messages can be categorized into system messages and user messages and both make up the prompt.

System messages are instrumental in shaping the model's behavior during the conversation. They serve as contextual instructions, guiding the model on how to generate its responses. For instance, a system message could instruct the model to mimic the speaking style of a specific character or adhere to a certain tone throughout the conversation. While system messages help tailor the model's responses, it is important to note that they do not

guarantee the model's behavior but merely influence it[2].

User messages, on the other hand, are the actual inputs provided by the user during the conversation. These could be questions, statements, or any other form of communication that the user wants the model to respond to. The model generates its responses based on these user messages, taking into account the context set by the system and prior messages [2].

The primary difference between system and user messages lies in their purpose and function. System messages guide the model's overall behavior and tone during the conversation, while user messages are the specific inputs that the model responds to. System messages can be seen as setting the stage for the conversation, while user messages are the actual conversation.

Max Tokens Parameter

Another important element in ChatGPT is the "max tokens" parameter. It is used to control the maximum length of the output text that the model generates. It essentially sets an upper limit on the number of tokens (words or parts of words) that can be in the generated response.

In ChatGPT it is a fixed value and it vary with the model selected, 4096 for if the model used is GPT-3.5-turbo or 8192 is the model selected is GPT-4 [32].

On the other hand with GPT models used via API it is possible to use the model providing a specific amount of max tokens response value, this is an advantage because allow the developer to choose a more tailored verbosity response provided by the model.

In terms of its specific value, it can vary based on the desired application or use case. If you want a shorter response, you would set a lower value. If a longer, more detailed response is needed, a higher value would be set. However, it's important to note that setting a very high value could lead to verbose outputs, and setting a very low value might cut off the generated text, leading to responses that may not make sense.

It's also worth noting that the total number of tokens in an API call both for input and output cannot exceed the model's maximum limit (e.g., 4096 tokens for gpt-3.5-turbo). This means that if your input message is very long, your output message could be cut short.

The exact "max tokens" value to use would depend on the specific requirements of the interaction or conversation you're designing. It's often a matter of trial and error to find the value that produces the best results for your specific needs.

2.3 Performance testing Tools

Performance testing is a subset of software testing that evaluates an application's performance under various workloads. It involves scrutinizing the user interface, generating test scripts, and comparing different metrics such as response time, CPU utilization, and memory allocation [8, 9].

In the context of modern software development and performance evaluation, performance testing tools play a pivotal role in assessing the capabilities, efficiency, and reliability of software systems. These tools provide invaluable insights by subjecting applications to controlled stress tests, measuring response times, and measuring their behavior under various loads and scenarios. Performance testing enables developers and system administrators to identify bottlenecks, performance gaps, and areas for optimization, thus contributing to the creation of robust and high-performing software solutions.

Performance testing tools

Performance testing tools are software applications or utilities designed to measure and evaluate the performance of various aspects of a computer system or software application. These tools are used to determine how well a system or application performs under specific conditions and compare it to other systems or known standards. The primary purpose of performance testing is to identify potential performance bottlenecks, optimize system performance, and make informed decisions about hardware or software upgrades.

There are several types of performance testing tools, each focusing on different aspects of performance evaluation: CPU, GPU, Memory, Network, Web Application and Database.

Performance testing software is an essential tool for evaluating the scalability, stability, and speed of software applications. It is used to simulate various load conditions and analyze the application's performance under these conditions.

There are many commercial and open-source performance testing tools available on the market, each with its own set of features, capabilities, and limitations. Some popular commercial tools include LoadRunner, Silk Performer, and NeoLoad, while popular open-source tools include Apache JMeter, Gatling, and Tsung. These tools can be used to test a wide range of applications, including web-based applications, databases, and web services.

Let go on to narrow it down to only the tools of Web Application performance testing tools.

Web Application performance testing tools

In this realm of performance testing tools, there exists a range of options, each with its own strengths and weaknesses. Noteworthy among these tools are Apache JMeter, LoadRunner, Microsoft Visual Studio (TFS), and Siegf, all of which are suitable for test automation.

Let's delve into more specific details about the capabilities of each performance testing tool following the research done by Abbas et al. (2017)[3]:

- Apache JMeter:
 1. Load Testing: Apache JMeter excels in load testing web applications and services, simulating multiple users and measuring how the system handles concurrent requests.
 2. Stress Testing: It can apply stress to the system by pushing it beyond normal capacity to evaluate its stability and responsiveness under extreme conditions.
 3. Protocols: JMeter supports various protocols, including HTTP, HTTPS, FTP, JDBC, SOAP, and more, allowing testing of diverse applications.
 4. Extensibility: JMeter is highly extensible through a wide range of plugins, enabling users to customize and enhance its functionality to suit specific testing requirements.
 5. Versatility: It can be used for small-scale as well as large-scale performance testing, making it suitable for a broad range of projects.
- LoadRunner:
 1. Enterprise-Level Testing: LoadRunner is designed for enterprise-level performance testing and can handle large-scale and complex applications with high concurrency.
 2. Protocol Support: It offers an extensive library of protocols, such as web, mobile, API, database, Citrix, and more, allowing comprehensive testing of various systems.
 3. Robustness and Scalability: LoadRunner is known for its robustness and ability to simulate a massive number of virtual users, providing accurate performance evaluation for critical applications.
 4. Diagnostics and Analysis: The tool provides integrated diagnostics and transaction tracing features to identify performance bottlenecks and root causes of issues.
 5. Cross-Platform Testing: LoadRunner supports performance testing across different platforms and technologies, making it suitable for diverse application landscapes.
- Microsoft Visual Studio (TFS):
 1. Integrated Load Testing: Visual Studio (TFS) offers integrated load testing features within the Visual Studio development environment, streamlining the testing process for developers.

2. **Web and Service Testing:** It specializes in performance testing for web applications, web services, and other types of applications, making it well-suited for web-centric projects.
 3. **Scripting Capabilities:** Visual Studio (TFS) supports scripting using C#, allowing developers to create and customize load testing scenarios to meet specific testing requirements.
 4. **Seamless Integration:** Since it is part of the Microsoft Visual Studio suite, it seamlessly integrates with other development and testing tools, providing a cohesive environment for software development.
- **Siege:**
 1. **Web Application Load Testing:** Siege focuses on HTTP and HTTPS load testing of web applications, making it a straightforward tool for basic web performance evaluation.
 2. **Simplicity:** Its command-line interface and easy setup make it user-friendly, particularly for those looking for a quick and simple load testing solution.
 3. **Unix and macOS Support:** Siege is compatible with Unix-like systems, such as Linux and macOS, making it ideal for testing on these platforms.

In summary, Siege stands out for its cost-effectiveness as an open-source tool, although it does have certain limitations. On the other hand, Microsoft Visual Studio (TFS) offers easy-to-use features and built-in testing capabilities. It excels at accommodating varying user requirements, whether for 100 parallel users or thousands. However, it is important to note that Microsoft Visual Studio (TFS) is limited to Windows OS and can entail high licensing costs. Meanwhile, Apache JMeter emerges as an optimal choice due to its cost-effectiveness, being freely available. Although it requires more time for initial installation, Apache JMeter boasts a wide array of options for result analysis and is well-suited for running different tests simultaneously. Additionally, its testing capabilities can be enhanced through the use of various plugins. Notably, comparative evaluations have shown that JMeter outperforms other tested tools or techniques, thanks to its ratio scale methodology and inclusion of consistency checks [3].

JMeter

Apache JMeter™ has emerged as a cornerstone in the realm of performance testing and evaluation. As an open-source Java application, JMeter was initially conceived to address the challenges of testing web applications, but its evolution has led to its adoption across a spectrum of testing functions. With its user-friendly interface and robust capabilities, JMeter enables software developers, quality assurance teams, and system administrators to comprehensively assess the performance and functional attributes of their applications [6].

At its core, JMeter excels in emulating real-world scenarios by simulating heavy loads on target components such as servers, networks, and endpoints. This capability proves invaluable in determining system resilience, scalability, and responsiveness under diverse and demanding conditions. Whether the focus is on web applications or other systems, JMeter stands ready to gauge the thresholds and capabilities of the software in question. Its utility extends beyond mere performance metrics; JMeter is adept at uncovering bottlenecks, identifying latency issues, and revealing potential vulnerabilities that might arise under high loads.

One of the hallmark features of JMeter is its ability to accommodate a wide range of applications, protocols, and platforms. From web technologies like HTTP and HTTPS to various programming languages like Java, NodeJS, PHP, and ASP.NET, JMeter provides a versatile testing ground. Furthermore, its adaptability is evidenced by its integration with popular continuous integration tools, such as Maven, Gradle, and Jenkins, allowing seamless inclusion within modern development pipelines [6].

Let's get into the architecture, methodologies and specific use cases of jmeter

Architecture and Methodologies

The architecture of Apache JMeter was designed with extensibility, modularity and flexibility in mind. It works using a decentralized architecture in which test plans are created and executed by the JMeter engine. This engine, written in Java, coordinates the execution of test scripts, manages threads, and collects results.

This design confers a significant advantage: one person can create a test and other people can run it to test the application in different configurations. There is no need to be a JMeter expert to run the tests. This means that the testing process can be easily scaled and distributed among a team, allowing each person to focus on his or her area of expertise.

JMeter Test Plan Structure

JMeter follows a methodology rooted in simulation. Test plans are crafted to replicate real-world user behavior, interactions, and load patterns [6].

A JMeter test plan describes a series of steps that JMeter will execute when run. It provides a framework for the test, including what to test and how to go about it. The test plan serves as the root node where all other elements like Thread Groups, Controllers, Samplers etc. are added.

Thread Group

In JMeter, a Thread Group is an essential part of any test plan. The Thread Group

represents a pool of virtual users that will execute the specific test plan [6]. Each thread simulates one real user making requests to the server. The controls for a thread group allow you to:

- Set the number of **threads**: This is the number of users JMeter will attempt to simulate.
- Set the **ramp-up** period: This is the time-frame over which JMeter will distribute the start of the threads.
- Set the number of **loop** iterations: This is the number of times the test will be executed by each simulated user.

For example, if you have 100 users and a 100-second ramp-up period, then the delay between starting users would be 1 second (100 seconds /100 users).

User Defined Variables

User Defined Variables in JMeter are a useful feature when you need to define your own variables and assign them certain values. They allow you to create default variables that can be used throughout your test plan. These variables are especially useful when you need to use the same value in multiple places in your test, and you want the ability to easily change that value [6].

JMeter also provides elements to generate random values, such as the Random Variable Config Element. This can be very useful when you need to simulate a more realistic scenario with varied input data. For example, you might need to simulate requests to a server with different user IDs or search terms.

These random values can also be assigned to User Defined Variables. So, instead of assigning a static value to a variable, you can assign a random value. This way, every time the variable is referenced, a new random value will be used. This can be done by combining the User Defined Variables element and the *__Random* function. For example, you could define a variable named "random_id" with the value "*__Random(1,100)*". This would generate a random number between 1 and 100 each time "random_id" is referenced.

For example, you might define a variable for the base URL of the website you're testing. Instead of typing the base URL into every HTTP Request sampler, you can use the variable instead, which makes your test plan easier to maintain and update.

Listeners

Listeners in JMeter are used to provide access to the data gathered by JMeter about the test cases during their execution. They allow users to view, save, and read the test results.

Listeners can be added at any place in the test, providing a clear and accurate picture of their performance [6].

There are many types of listeners available in JMeter, each providing a different way to view the collected data. For instance, the View Results Tree listener shows a tree of all sample responses, allowing you to inspect every single request and response in detail. The Summary Report listener provides a table with aggregated metrics about all requests, such as the average response time, min and max response time, error percentage, and more.

Practical Use Cases

The practical use cases of Apache JMeter are diverse and span across multiple domains:

- **Web Application Testing:** JMeter excels in testing web applications by emulating user interactions, measuring response times, and identifying bottlenecks. It enables load testing, stress testing, and capacity planning, ensuring that web applications perform optimally under various user loads.
- **API and Web Services Testing:** JMeter's versatility extends to API testing, where it can send various types of requests like HTTP, SOAP, REST, and more. It validates the responses, checks data accuracy, and assesses the reliability of APIs.
- **Database Testing:** JMeter can simulate database loads by sending SQL queries, measuring database performance, and ensuring the database can handle concurrent user requests.
- **Performance Testing:** JMeter facilitates performance testing by subjecting applications to different load levels, helping identify thresholds, scalability issues, and response time variations.
- **Security Testing:** With JMeter's ability to generate a wide range of requests, it is used for security testing by sending malicious requests and assessing an application's resilience against potential attacks.
- **Continuous Integration:** JMeter integrates seamlessly into CI/CD pipelines, enabling automated performance testing as part of the development process. This ensures that performance issues are caught early in the development lifecycle.

2.4 Related Works

This section provides an in-depth review of existing literature related to three key areas pertinent to this thesis: DevOps, Generative AI, and the use of Artificial Intelligence in Performance Testing.

DevOps Related Works

Now let's look more specifically at the benefits of using DevOps for software development.

DevOps combines development (Dev) and operations (Ops) to unite people, process, and technology in application planning, development, delivery, and operations [18]. When a team adopts DevOps culture, practices, and tools, they can achieve remarkable outcomes:

- **Accelerate time to market:** Through increased efficiencies, improved team collaboration, automation tools, and continuous deployment, teams are able to rapidly reduce the time from product inception to market launch.
- **Adapt to the market and competition:** A DevOps culture demands teams to have a customer-first focus. By combining agility, team collaboration, and a focus on the customer experience, teams can continuously deliver value to their customers and increase their competitiveness in the marketplace.
- **Maintain system stability and reliability:** By adopting continuous improvement practices, teams are able to build increased stability and reliability into the products and services they deploy. These practices help reduce failures and risk.
- **Improve the mean time to recovery:** The mean time to recovery metric indicates how long it takes to recover from a failure or breach. To effectively manage software failures, security breaches, and continuous improvement plans, teams should measure and work to improve this metric.

DevOps practices also improve organizational culture and enhance employee engagement.

Recent studies show that implementing DevOps can lead to a 24x faster recovery time and high-performing organizations spend 22 percent less time on unplanned work and rework. Consequently, they are able to allocate 29 percent more time to new work, such as new features or code [20]. The primary driver of this improvement is the implementation of CI/CD pipelines.

CI/CD pipelines contribute to development improvement by:

- **Enabling faster feedback:** CI/CD allows developers to quickly receive feedback on code changes, helping them identify and address issues early in the development process.
- **Ensuring code quality:** By automating the build and testing processes, CI/CD pipelines enforce code quality standards and reduce the risk of introducing bugs and errors.

- Facilitating frequent releases: CI/CD pipelines automate the deployment process, making it easier to release new features and updates more frequently, resulting in faster time-to-market.
- Promoting collaboration: CI/CD encourages collaboration between development, testing, and operations teams, fostering a culture of shared responsibility and accountability.

This integration of CI/CD pipelines with DevOps practices leads to improved development outcomes and helps organizations achieve their goals more effectively.

Generative AI Related Works

When discussing Generative AI in the realm of code creation and suggestion, it is crucial to mention the collaboration between GitHub and OpenAI, resulting in the development of GitHub Copilot. This model, trained on vast amounts of code, leverages natural language prompts to provide coding suggestions across multiple programming languages [1]. Extensive research has indicated that GitHub Copilot enables developers to code more efficiently, focus on tackling larger problems, remain in a state of flow for longer periods, and experience greater fulfillment in their work [19]. Moreover, empirical studies have demonstrated how GitHub Copilot effectively strengthens pair programming relationships among developers, spanning various programming languages [16, 28].

While GitHub Copilot proves to be an invaluable tool for developers, it is important to acknowledge that it has certain limitations in terms of code optimization and natural language understanding. Consequently, it is advisable for GitHub Copilot to be utilized by experienced programmers or individuals with a solid understanding of the specific context they are working within [26].

OpenAI claims that GPT-4 outperforms previous large language models across various domains, including code generation, logical reasoning, and mathematical reasoning. Furthermore, the latest model demonstrates an 82% reduction in the likelihood of providing disallowed content and a 40% increase in the likelihood of producing factual results compared to its predecessor, addressing concerns about misinformation [31].

One notable advancement in the latest model is the increased context length. GPT-4 introduces two variants with 8,192 tokens and 32,768 tokens, which is approximately eight times longer than GPT-3.5, allowing users to process larger volumes of text and perform tasks such as long document summarization. Both versions have been trained on data up to September 2021, similar to the training data of previous models [31].

Furthermore, a significant breakthrough with GPT-4 is its ability to accept images as

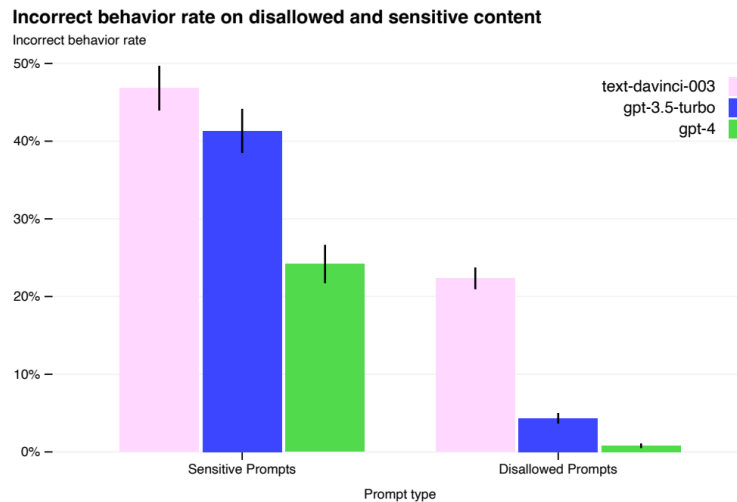


Figure 2.3: Rate of incorrect behavior on sensitive and disallowed prompts. Lower values are better. GPT-4 RLHF exhibits significantly lower incorrect behavior rates compared to prior models [32].

inputs. This means that visual data, such as photographs or physical data, can be directly processed using the model, enabling the generation of insights, summaries, Q&A, and more [31].

Last but not least, GPT-4 excels in terms of safety and security of output prompts, as depicted in Figure 2.3.

OpenAI’s safety mitigations have significantly enhanced many of GPT-4’s safety properties compared to GPT-3.5. The model demonstrates an 82% decrease in its tendency to respond to requests for disallowed content compared to GPT-3.5, and it adheres to policies regarding sensitive requests (e.g., medical advice and self-harm) 29% more frequently [31].

Studies such as ”CAN ARTIFICIAL INFLUENCE TRANSFORM DEVOPS?” by Alenezi, Zarour, and Akour (2022)[4] and ”How artificial intelligence (ai) and machine learning are changing devops” by Kinsbruner (2020)[21] delve into the transformative potential of AI, exploring how it can augment the functionality of DevOps. These researchers elucidate how AI can automate critical DevOps processes such as code reviews, streamline team operations, and enhance problem-solving capabilities.

Another pivotal concept that has garnered significant attention is Machine Learning DevOps (ML-DevOps), often misconstrued due to several myths prevalent in the literature. Scholars such as Poloskei (2022)[34], Tamburri (2020)[39], and Treveil (2020)[40] have undertaken comprehensive analyses of ML-DevOps from an end-to-end perspective. They have contextualized ML-DevOps as a specialized subset of DevOps, located at the intersection of

software engineering, data engineering, and Machine Learning. This clarity in understanding is furthered by studies like those conducted by Alla (2021)[5], Gift and Deza (2021)[15], and Symeonidis et al. (2022)[38], who have strived to define ML-DevOps with precision, contributing to a common understanding of the paradigm.

Parallely, the emergence of DevSecOps and AIOps, as extensions of DevOps, has been a significant focus in the realm of software development and IT operations. DevSecOps aims to identify and mitigate application vulnerabilities early in the development process, a concept discussed in detail by Rajkumar et al. (2016)[35]. On the other hand, AIOps marks the integration of AI and Machine Learning into the DevOps life cycle, enhancing various IT tasks such as data monitoring, predicting the effects of deployment, and automatically responding to environmental changes, as elaborated by Ebert et al. (2016)[11].

Performance Testing Related Works

Moreover, AI and Machine Learning techniques are increasingly being employed in the development and testing processes of these applications. Specifically, Performance Testing (a crucial facet of software testing) is experiencing a significant impact of AI.

AI's Contribution to Performance Testing

The comprehensive performance testing process, which necessitates continuous manual inputs and in-depth testing, poses several challenges. These include the time-intensive scripting of real user experiences, analysis of extensive data, identification and elimination of performance bottlenecks, manual generation of test cases, and manual root cause analysis of failures [9].

AI can address these challenges by automating nearly every stage of the application performance testing phase. The outcome is a user-friendly product that reaches the consumer in less time, utilizing fewer resources and manpower [8, 9].

Benefits of AI-Driven Performance Testing

The integration of AI into performance testing presents numerous advantages:

- **Precise Data Analysis and Delivery:** AI-based testing techniques outperform manual testing in terms of accuracy and data delivery. AI minimizes the potential for human error, thereby enhancing the precision in product development [9].
- **Time and Cost Efficiency:** By automating testing tasks, AI significantly reduces the time taken and eliminates overhead costs compared to manual inputs [8, 9].

- **In-depth Error Detection:** AI's capacity to detect errors and discrepancies surpasses human capabilities, simplifying the tracing of bugs and defects that might be missed by manual and conventional testing methods [9].
- **Pattern Recognition and Visual Testing:** AI can identify and rectify visual bugs, ensuring all graphic components operate as intended, thereby adding flexibility to software performance testing [8].
- **Error-Free API Generation:** AI-enabled performance testing tools can generate and verify whether an API (Application Programming Interface) is functioning as per instructions, ensuring seamless communication between API tools and databases [8, 10].
- **High-Quality Test Cases:** AI facilitates the generation of superior quality test cases. Unlike developers using traditional methods, those leveraging AI can evaluate numerous test case scenarios and implement innovative methods to test cases within seconds [9].

AI Tools for Performance Testing

The market offers several AI tools for performance testing. Some of the notable ones include:

- **Mabl:** A popular DevTestOps platform, Mabl simplifies the creation and automation of user interface tests. It offers unlimited testing and scaling scripts, simultaneous testing, a secure cloud platform, and self-correcting test scripts [8].
- **TestCraft:** Known for reliable testing when paired with Selenium, TestCraft is an AI-powered performance testing tool. It generates a test model of the actual scenario, facilitating easy code editing [8].
- **Applitools:** A prominent AI-enabled visual apps management testing and monitoring tool, Applitools offers an end-to-end, intelligent performance testing platform preferred by software development professionals [8].
- **Functionize:** A cloud-based, AI testing technology, Functionize is renowned for load and performance testing. It uses AI and Machine Learning techniques to enhance test script creation, debugging, and script maintenance. It also incorporates Natural Language Processing (NLP), enabling it to understand commands in plain English while creating functional test scripts [8].

AI automation has simplified the testing process beyond the capabilities of manual testing. It's crucial to choose the right combination of tools suitable for the product, create code-less

automated scripts, and always keep the user's perspective in mind while testing. The integration of AI in performance testing is revolutionizing the way businesses develop their software products by delivering high-quality solutions in less time and at a lower cost. The use of AI for performance testing is anticipated to gain widespread popularity in the coming years [8, 9, 10].

Chapter 3

DevOptimizeAI Requirements and Design

This chapter provides a comprehensive exploration of the essential requirements and design considerations for DevOptimizeAI.

The mission section provide a description of the mission of DevOptimizeAI and describe some use cases.

The requirements section details the functional and non-functional requirements that guide the DevOptimizeAI development. This section ensures DevOptimize is tailored to meet the specific needs of users.

The design section presents the DevOptimizeAI design,from high to low level design.

3.1 DevOptimizeAI mission

DevOptimizeAI's mission is to improve the efficiency and quality of Azure Functions software by integrating the performance testing capabilities of Apache JMeter along with OpenAI's GPT models into existing DevOps pipelines.

Azure Functions are serverless computing resources provided by Microsoft's Azure cloud platform. They allow developers to run code in response to various events without managing the underlying infrastructure.

The uniqueness of DevOptimizeAI lies not within its individual components but in their synergy. The objective is to ensure the efficiency of an application, even if it's developed by multiple hands or with different knowledge background.

DevOptimizeAI achieves this by analyzing Azure Function code in conjunction with the execution and analysis of performance tests.

This analysis produce a detailed report as shown in the Appendixes E, F, G and H about the code describing how to improve the performance and how to improve the overall written code, also in terms of quality of code, also proposing better ready to use implementation of the code.

This empowers developers at all levels of experience, from junior to senior, by assisting them in optimizing their code, identifying potential performance bottlenecks, and improving the overall software development process.

The strength of DevOptimizeAI lies in its ability to assist multiple developers working on the same project simultaneously, even if they do not have any knowledge about performance testing and prompt engineering. Furthermore, it can do so automatically following a single configuration, making it a powerful tool for enhancing the software development process in an Azure Function and Azure DevOps development environment.

DevOptimizeAI's mission goes beyond empowering developers; it aims to contribute to the production of high-quality, efficient software. To better define this mission, we present a practical use case example followed by a number of possible more general uses of DevOptimizeAI

Case 1: Scenario without DevOptimizeAI

Suppose Alice and Bob are senior and junior developers, respectively, and are working jointly on a project using Azure DevOps to manage it. Alice instructs Bob to create the connection to a database for a set of REST APIs that will be managed by an Azure Function. Bob creates the connection, brings the changes into the test environment, and asserts, after a series of his own tests, that it works. So Bob brings the updates into production. Alice unfortunately cannot validate the work done by Bob as she has other urgent tasks to handle. When Alice sees the code written by Bob, given her experience, she tests it with tests she wrote on JMeter to monitor performance and modifies Bob's code to make it more efficient. DB calls have been inefficient for two weeks, Bob has brought an inefficient version of the project into production, and Alice has had to waste valuable time for her to modify Bob's version of the connection.

Case 2: Scenario with DevOptimizeAI

If we assume instead that there are always Alice and Bob, but that they use DevOptimizeAI then the scenario looks rosier. Alice produces the tests with JMeter while Bob produces the connection to DB. When Bob brings the changes into the test environment after creating the connection, DevOptimizeAI will perform the analysis and improvement guidance for the code written by Bob.

The beauty of this scenario is that Bob doesn't need to be familiar with JMeter or

performance testing and also he doesn't need to be expert in prompt engineering and GPT models.

DevOptimizeAI does the complex work, generating actionable advice that Bob can follow to enhance his code. This way Bob can modify the code by improving it and bring the most efficient version into production.

This scenario presents a win-win situation for both Alice and Bob. For Alice, validation becomes a breeze as she can rely on DevOptimizeAI to assess Bob's work for efficiency. For Bob, the benefit is twofold: he is able to write more efficient code and learn from the advice provided by DevOptimizeAI. In essence, DevOptimizeAI not only promotes efficient and superior code but also fosters a learning environment, contributing to the professional growth of developers like Bob.

There are some other examples of how the "DevOptimizeAI" prototype could be used:

1. Performance Testing: A company wants to ensure its Azure Functions can handle high user loads. They use DevOptimizeAI to conduct performance testing with JMeter, identifying potential performance issues with the GPT model. This performance testing is also enhanced by the fact that is conducted an analysis of the test results providing further insight about the output values of the test (analysing execution time, throughput and percentage of errors).
2. Code Quality Improvement: A novice developer is learning best coding practices. They use DevOptimizeAI to receive feedback and guidance on their Azure Functions code, helping them understand how to improve the efficiency and quality of their coding.
3. Bottleneck Identification: A team is facing performance issues with their software but can't identify the bottleneck. By integrating DevOptimizeAI into their DevOps pipeline, they receive insightful feedback pinpointing the potential bottlenecks and suggestions for improvements.
4. Learning and Training: An educational institution uses DevOptimizeAI as a teaching tool in their software development courses. Students can get immediate feedback on their coding assignments, helping them learn and improve their coding skills.

3.2 Requirements

This section outlines the essential requirements for DevOptimizeAI, categorized into functional and non-functional.

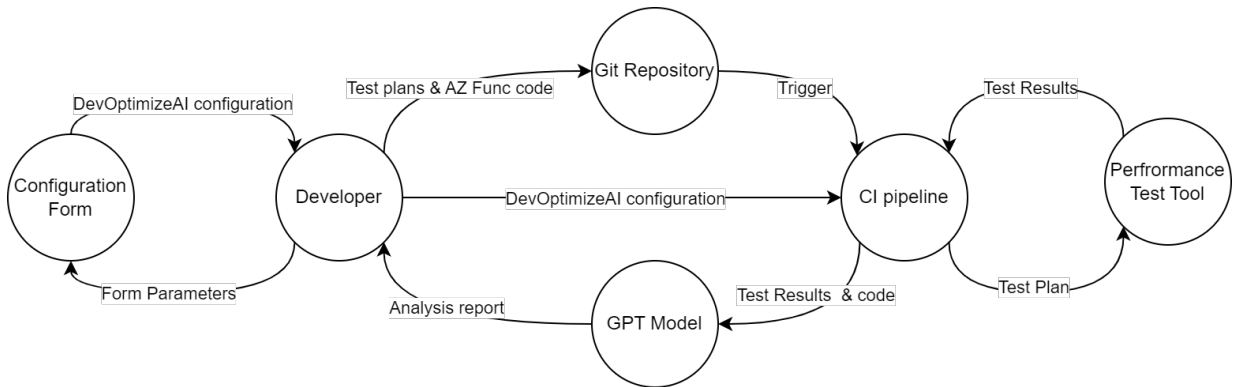


Figure 3.1: Data flow diagram

Functional Requirements

The following subsection presents the functional requirements of the DevOptimizeAI system, outlining the specific responsibilities and expected behaviors of each component in the system.

The functional requirements are also visually depicted in an Entity-Relationship Diagram (ERD) in Figure 3.2. The ERD provides a clear and concise visual representation of the interactions and relationships between the different entities involved in the system. In addition, the Sequence Diagram in Figure 3.3 offers an in-depth illustration of the sequence of actions performed during the utilization of DevOptimizeAI, including the interaction among the various components over time.

A comprehensive view of the data movement between the components is provided in a Data-Flow Diagram (DFD) shown in Figure 3.1. This diagram offers a perspective on how data flows and is processed within the system, contributing to a complete understanding of the system's operation.

The involved entities in the system include the Developer, the Configuration Form, the Azure Function Code and Test plan, the Git Repository, the CI Pipeline, the Performance Test, the GPT model, the Test Result, and the Analysis Report. Each of these entities has specific roles and responsibilities in the system, as detailed below.

- **Developer:** The Developer should be able to create one or multiple test plan and write one or multiple code. Both, code and test plan should be correct and do not contain errors. However, in certain scenarios, a developer may not possess the knowledge or resources to generate a test plan. In such cases, they can leverage test plans produced by other developers in the team. This allows for greater collaboration and knowledge sharing, as well as ensuring the efficiency and correctness of the code

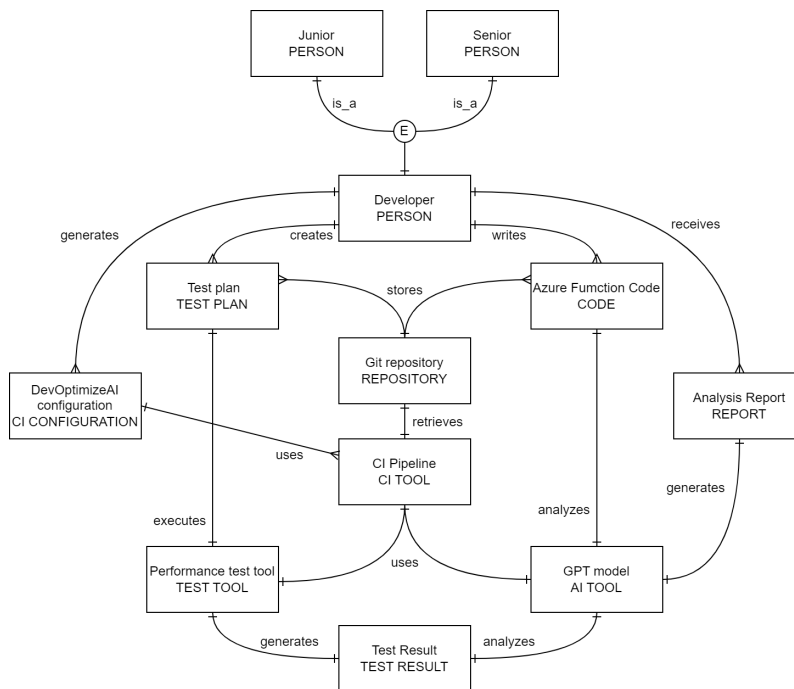


Figure 3.2: Entity-Relationship requirements diagram

without requiring every developer to have expertise in test plan creation.

The Developer should also know the DevOps paradigm. The Developer should also be able to receives and understand the analysis report.

- **Configuration Form:** DevOptimizeAI should provide an easy-to-use form for developers to fill out, which will generate the necessary configuration that should be cut&pasted to the pipeline for the performance test and GPT model analysis.
- **Azure Function Code and Test plan:** Both Azure Function code and test plan should be submitted by the developer into the same Git repository. Both should be without syntax errors and the code must be able to compile and the test plan should be able to be exetuted by JMeter . The test plan should the accessible to the performance test tool and the code should be accessible to the GPT model.
- **Git Repository:** The system should be able to integrate with the Git repository that hosts multiple code and test plan.
- **CI Pipeline:** Upon submission of the project or a change done to the Git repository, the CI pipeline process should be able to automatically start. The CI pipeline should be able to use the performance test tool and the GPT model and should be able to integrate these components together. The CI Pipeline should also:

1. Improve the results provided by the JMeter test by going to format them so that they are more understandable to the GPT model.
 2. Configure of a robust and detailed prompt message for the GPT model.
 3. Delivery of the analysis report to the developer team.
- **Performance Test:** DevOptimizeAI should be able to use the performance testing tool and should be started by the CI pipeline and execute the test plan stored in the Git repository. After the execution it should generate the test result in a format that could be analyzed by the GPT model with the code.
 - **Test Result:** The test result should be generated by the performance test tool and analyzed by the GPT model with the code to generate the analysis report.
 - **GPT model:** The DevOptimizeAI GPT model should be used directly by the CI pipeline. The GPT model should be able to analyze a test result and a code simultaneously to generate an analysis report.
 - **Analysis Report:** Analysis report should be produced and provided securely and quickly to the Developer. It should be readable and well formatted and provide useful insight to the developer.

Sequence diagram

To provide a better comprehension of the interaction between elements in the system there is provided a sequence diagram in Figure 3.3.

This sequence diagram communicates the flow of operations in the CI environment, highlighting the sequential and automated nature of the tasks following a code commit action by the Developer, in particular:

1. The process initiates with the Developer filling out a Configuration Form. The purpose of this form is to gather the necessary details for the upcoming stages.
2. The Configuration Form, upon receiving the input, generates a YAML script and returns it to the Developer.

The Developer then:

- 3.1 adds this generated YAML script into the CI Pipeline
 - 3.2 commits the Azure Function Code and JMeter file (.jmx) to the Git Repository.
3. The action of committing triggers the CI Pipeline, and a series of automated tasks are set into motion.

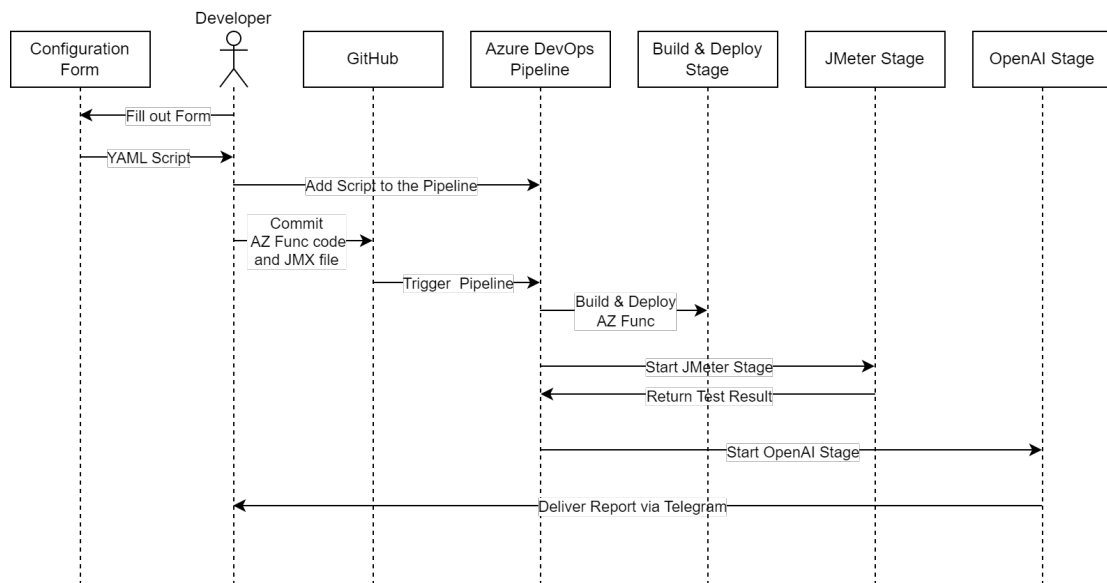


Figure 3.3: High level sequence diagram

4. The first task from the CI Pipeline involves the Build & Deploy Stage. Here, the Azure Function code that was committed by the Developer is built and deployed.

Once the Build & Deploy Stage is complete, the CI Pipeline starts the JMeter Stage in which:

- 6.1 JMeter executes the test plan outlined in the JMX file.
- 6.2 returns the test log file summarized to the CI Pipeline.

With the completion of the JMeter Stage, the CI Pipeline triggers the OpenAI Stage in which:

- 7.1 A GPT model analyse the test result and the Azure Function code.
- 7.2 A Python script prepare and send the analysis report to the developer via Telegram.

Non-Functional Requirements

Quality Requirements:

- System Performance: DevOptimizeAI should be capable of handling varying sizes of code but smaller than the maximum input token available for the model chosen

for the analysis (e.g. GPT-4 could handle about 32k token, the equivalent of about 100k characters) and any numbers of developers, queueing the request, without compromising on performance.

- **Analysis Speed:** The GPT model should complete the analysis of code and test results in a from 2 up to 5 minutes, to not delay the development process.
- **Scalability:** DevOptimizeAI should be designed to maintain performance as the load increases, supporting an increasing number of developers, handling larger codes and more complex test plans.
- **Usability:** DevOptimizeAI should provide an interactive and user-friendly interface, making it easy for developers of all experience levels to interact with DevOptimizeAI and understand the analysis reports.
- **Reliability:** DevOptimizeAI should consistently produce accurate and reliable analysis reports, contributing to the production of high-quality, efficient software.

External Interface Requirements:

- **Interoperability:** DevOptimizeAI should seamlessly integrate with existing CI pipeline tools, specifically it should support various Git repository platforms and be able to effectively work with performance testing tools for performance testing.
- **User Interface:** The analysis report should be presented in a clear and understandable manner, accommodating developers at all experience levels.

3.3 Design

This section discusses the DevOptimizeAI design. The design process is guided by the functional and non-functional requirements outlined in the previous section. The design process takes inspiration from similar implementations in prior research [33], with certain modifications to better suit the requirements of this thesis.

High-level design

The software design specification is divided into three main part: system overview design, high-level pipeline design and high-level DevOptimizeAI design. The creation of the high-level design was done from an architectural design proposed by Microsoft, specifically: CI/CD baseline architecture with Azure Pipelines [17].

System overview design

The system overview design will present a comprehensive view of the system environment. This includes all the external entities that interact with our system, such as the Developer, Git Repository, and the configuration form. The architecture in Figure 3.4 provides a broad perspective of how the system fits within its environment and communicates with external interfaces.

In system overview design are represented:

- **Developer:** The developer is also the end-user and it must provide JMX file and Azure Function code into a GitHub repository. The developer should also use the configuration form to configure the pipeline stage and subsequently add that to the project pipeline. At the end of the pipeline the developer should receive the analysis report from the Azure DevOps pipeline
- **Configuration form:** The configuration form should be implemented via Web app to provide a very simple, lightweight and user friendly form from which the developer could cut&paste the YAML configuration script.
- **YAML configuration script:** The YAML configuration script is the output of the configuration form. The developer should add this script in its pipeline to empower the deployment of its Azure Function. This script adds at the pipeline the DevOptimizeAI stages.
- **JMX file:** A JMX file is an XML file used by Apache JMeter to store test plans. JMeter's test plans are the cornerstone of any JMeter test, as they describe a series of steps JMeter will execute when run.

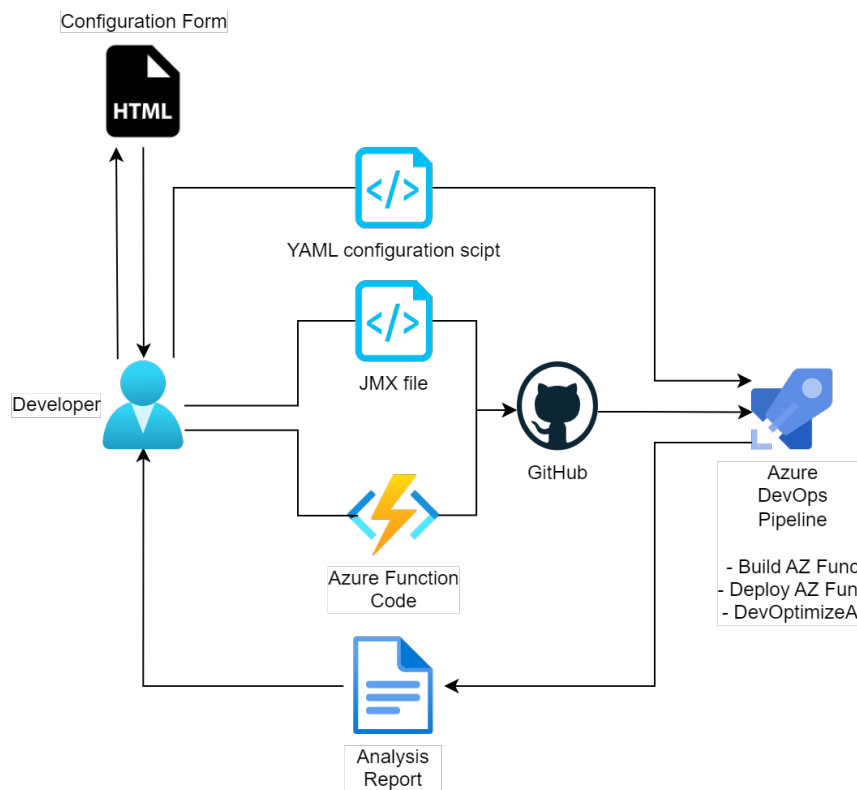


Figure 3.4: Architecture - system overview

- **Azure Function code:** Azure Functions is a serverless compute service provided by Microsoft Azure. It lets developers run small pieces of code (called "functions") in the cloud without needing to provision or manage infrastructure. The code for Azure Functions can be written in several programming languages like C#, Java, JavaScript, TypeScript, and Python. The code is event-driven, meaning it runs in response to triggers like HTTP requests, database operations, queue messages, or timers.
- **GitHub:** GitHub is a web-based platform used for version control and source code management. GitHub allows developers to collaborate on projects and keep track of changes to any set of files. Its features include code reviewing, project managing, integrating with other platforms, hosting documentation, and team management.
- **Analysis report:** This is a markdown file that will contain explanation of the test results and suggestion to improve the code. In particular this could include performance testing results, security analysis, code review, or any other type of software analysis. The report typically includes information on the methods used for the analysis, the results, and any recommendations or steps for improvement.
- **Azure DevOps Pipeline:** Azure DevOps Pipeline is a cloud service provided by

Microsoft Azure to automatically build, test, and deploy your code to any platform. It integrates perfectly with GitHub, making it easy to implement CI/CD practices. Pipelines can be configured through a web interface or a YAML file, and they support a wide range of languages, platforms, and cloud services. In pipeline must be present a build and deploy stage in which will be provided the deployment of the Azure Function before the start of the DevOptimizeAI stage.

High level pipeline design

Moving to high level pipeline design the focus move on its major components and illustrating how they interact with each other and the external entities identified in system overview. This will give a more detailed view of the Pipeline's internal structure and the interactions between its key components.

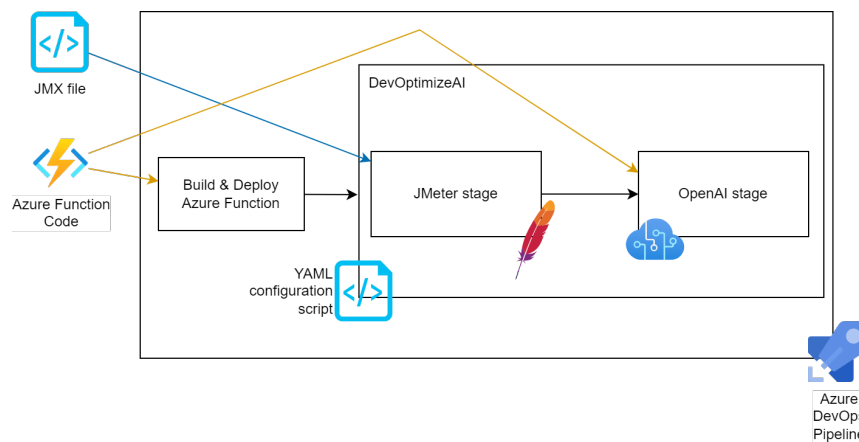


Figure 3.5: High-level pipeline design

High-level pipeline design is show in Figure 3.5 and it represents:

- **AZ Func code and JMX file:** Here are shown the previously presented Azure Function code and JMX file. This diagram shows the pipeline stages where these components are used. The JMX file is used in the JMeter stage, while the Azure Function Code is used in the Build and Deploy stages and the OpenAI stage.
- **Build and Deploy stage:** This is the initial stage where the code is compiled and built. If the build is successful, it is then deployed to the specified environment. This stage may involves processes like code compilation, unit testing and deployment. These two stage should be done by the Developer configuring the deployment of the Azure Function.
- From the cut&pasted YAML configuration script are then provided two more stages:

- **JMeter Stage:** In this stage, the deployed application is tested for its performance using Apache JMeter. JMeter simulates different user loads and behaviors as configured via the JMX file and the DevOptimizeAI configuration and tests how the application performs under these conditions. The results from these tests are then collected and prepared for the next stage.
- **OpenAI Stage:** Here, the results from the JMeter Stage are analyzed using OpenAI. This AI model can understand the test results, identify patterns, detect potential bottlenecks or areas of improvement in the Azure Function code, and provide suggestions for optimization. The output of this stage is a detailed analysis report that provides insights into the performance and efficiency of the code.

High level DevOptimizeAI design

High-level DevOptimizeAI design, finally, delve into the specific details of the DevOptimizeAI pipeline part. This level dissects the main components into their individual parts, providing a deep understanding of the inner workings of the DevOptimizeAI pipeline. This includes the processes and functionalities that enable the system to perform its tasks.

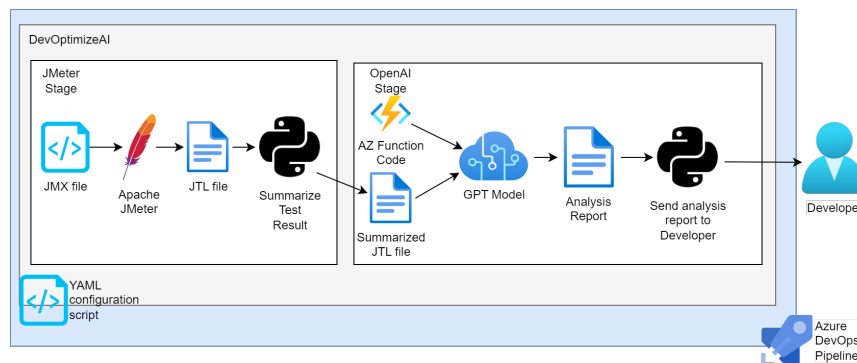


Figure 3.6: High-level DevOptimizeAI design

High-level DevOptimizeAI is shown in Figure 3.6 and it represents:

- In JMeter stage:
 - **JMX file:** This is the previously described JMX file submitted by the developer. In this phase is read from the GitHub and passed to the Apache JMeter.
 - **Apache JMeter:** Now Apache JMeter execute the test plan provided and produce the JTL file.

- **JTL File:** JTL is a file format used by JMeter to log test results. The results can then be displayed by the various JMeter Viewers, or processed externally with further analysis.
- **Summarize Test Result Script:** This is a script that parses the JTL file and summarizes the test results, providing key metrics and insights about the performance tests keeping the size of the file limited to permit a better future analysis.
- In OpenAI stage:
 - **Azure Function Code:** This is the code of the previously built and deployed function app. It will be part of the inputs for the GPT model analysis
 - **Summarized JTL Result:** This is the output of the previous stage of the pipeline.
 - **GPT Model:** A machine learning model that analyzes the JTL results and the Azure Function Code. It will be prepared with the appropriate prompt to generate tailored analysis report. It is prepared by packaging the JTL results and the Azure Function code into system and user messages. The system message serves as a high-level directive, informing the GPT model about the task at hand (the analysis of JTL results and Azure Function code). The user message provides the actual input data for the task, i.e., the JTL results and the Azure Function code.
 - **Analysis Report:** This report contains the output from the GPT model. It provides insights into the performance of the tested application and suggests areas for improvement. It is in markdown format to a better visualization of the code and text parts that will produce.
 - **Send Analysis Report To Developer:** This is the final step where the analysis report is sent to the developer, via Telegram, so they can review and act on it. The use of Telegram for delivering these insights has several advantages over traditional methods like email, for example the platform allows for easy group collaboration as well as safer and faster messages.

Low-Level Design

In the low-level design phase of the system the three key components will be analyzed: the configuration form, the JMeter stage, and the OpenAI stage. Each of these components will be broken down into their finer details, illustrating the implementation specifics, interaction pathways, and data structures.

The **configuration form** is an integral part of the user interface, designed to gather necessary information from the developers. This stage of the design will detail the layout, field specifications, and data validation rules. Additionally, it will outline the process of form submission and the communication flow to the next stage.

The **JMeter stage** is responsible for performance testing of the deployed application. In the low-level design, we'll outline the specific utilization of Apache JMeter, including the structure of the JMX file it uses, the test execution process, and the format of the JTL file it produces to log test results.

The **OpenAI stage** employs artificial intelligence to analyze test results. In this part of the design, we'll explain the specifics of the AI model used, how it interprets the JTL results, and how it generates and deliver an analysis report.

Configuration form

The DevOptimizeAI configuration form will be implemented as a web application, is the fundamental user interface to allow Developers to integrate DevOptimizeAI within their pipeline. It will need to be lightweight and portable as well as intuitive. It will need to allow the configuration of several parameters useful for both the JMeter stage and the OpenAI stage. These parameters will need to be:

- For the JMeter stage:
 - **Threads:** This parameter determines the number of virtual users or threads that JMeter will simulate to put a load on the server. This is crucial for understanding how the system performs under various loads.
 - **Rampup:** This parameter is used to specify the time taken to create the total number of threads for the test execution. It aids in gradually increasing the server load to assess its performance.
 - **Loops:** This parameter indicates the number of times the test case needs to be executed. It allows for repeated execution to ensure the system's stability over time.
- For OpenAI stage:
 - **GPT Model:** The choice should be limited between GPT 3.5 and GPT 4.
 - **Max Token:** This parameter limits the maximum number of tokens in the generated output. It helps control the verbosity of the model's response. Higher Max token could provide more detailed response, but if this value is too high could cause error due to the model inability to deal with too large texts.

- **Temperature:** This parameter, bounded between 0 and 1, controls the randomness of the model’s output, affecting the diversity of the generated responses. For instance lower values like 0.1 would make the model’s outputs more deterministic, the output will be ”safer” and more focused, but it may lack diversity and creativity. On the other hand, higher values like 0.9 increase randomness this can also result in more mistakes or nonsensical outputs.
 - **Message caption:** This parameter allows for the input of a custom caption that will be included in the message containing the Analysis report. This will allow developer to better discriminate different tests.
 - **chatid:** This parameter represents the unique identifier of the Telegram chat where the AI’s analysis report will be sent.
- For Pipeline configuration:
 - **Root Directory:** This parameter specifies the root directory where all the necessary files for both stages are stored.
 - **Working Directory:** This parameter points to the active directory where the Azure Function code is stored. The combination of Root Directory and Working Directory provide full path to the Azure function code.
 - **Service Connection to azure keyvault:** This parameter allows for the secure storage and retrieval of the secrets related to the GPT model used, ensuring a secure connection to Azure Key Vault.

All these parameters will be aggregated in order to produce a configuration text for connecting to the DevOptimizeAI pipeline in a simple and intuitive way.

JMeter stage

The JMeter stage is the first stage in the DevOptimizeAI pipeline. In this stage, performance testing is performed with respect to the previously deployed Azure Function. This stage is triggered from the developer pipeline with the previously described YAML configuration script. This pipeline stage will need to:

- **Read the JMX file:** In this step, the JMX file, which contains the test plan, is read. The parameters for threads, rampup, and loops are then set up based on the data retrieved from the previous configuration file. This ensures that the JMeter stage aligns perfectly with the developer’s initial configuration, leading to more accurate and relevant performance tests.
- **Execute the test plan:** Based on the details read from the JMX file, the test plan is executed. This includes simulating the specified number of users, requests, and

other configurations to test the application's performance under various loads. This is the main part of this stage and the one that is more time consuming depending on the test that should be carried out.

- **Produce the JTL file:** Upon completion of the test plan execution, the results are compiled and logged into a JTL file. This file includes crucial data such as response times, error rates, and throughput, which will be used in the next stage for analysis.
- **Summarize the Test Result:** After the JTL file has been produced, a summary of the test results is created. This summary provides a quick overview of the test execution, highlighting any significant findings or anomalies that occurred during the testing. This is mandatory due to the fact that for larger test simulation the results could produce large JTL file making the GPT Model goes in error.

By providing the model with a condensed yet comprehensive summary instead of the complete JTL file, we not only prevent these potential issues, but also enable the model to focus on the most relevant data. This leads to more accurate, meaningful, and appropriate responses from the model, enhancing the overall quality of insights generated and ensuring the reliability of the outcomes.

OpenAI stage

The OpenAI stage is the last in the DevOptimizeAI Pipeline and the one with the most tasks. During this stage the pipeline will have to:

- **Read the Azure Function code and the summarized test file:** This task involves reading and interpreting the Azure Function code and the summarized test result file generated from the JMeter stage. This data forms the basis for the subsequent analysis by the GPT Model.
- **Call the GPT Model with engineered prompt:** In this task, the GPT Model is invoked with an appropriately crafted prompt. This prompt is engineered based on the Azure Function code and the summarized test results, and is designed to guide the GPT Model to provide the most relevant analysis and suggestions. By providing a well-formulated prompt, the model is given clear direction regarding the context and the desired type of response. This not only streamlines the model's processing but also significantly enhances the relevance and accuracy of the outputs.
- **Generation of the Analysis report:** The GPT Model generates an analysis report based on the interpreted test results and the Azure Function code. This report contains valuable insights about the performance of the Azure Function, including potential bottlenecks and areas for optimization.

- **Deliver the Analysis report to the developer via Telegram:** Once the analysis report is generated, it is delivered to the developer through a Telegram message. This provides a convenient and immediate way for the developer to receive the insights generated by the OpenAI stage.

This part of the design assume that the Developer know how to configure a receiver bot on Telegram.

Chapter 4

DevOptimizeAI Implementation

The chapter delves into the details of the Configuration Form, breaking down its structure, functionality, and role in the system. This will highlight how this component facilitates the customisation and control of the testing process.

The chapter will also cover the specifics of the DevOptimizeAI pipeline stages, the JMeter stage and the OpenAI stage. This will provide insights into how these stages work together to run tests and generate useful feedback.

Finally, it will explore the GPT response formatting, explaining the different types of responses that the model can provide and their significance in the context of DevOptimizeAI.

4.1 Configuration Form

The Configuration form Web App provides a user interface for configuring parameters related to OpenAI and JMeter Test and it is shown in Figure 4.1.

It's important to note that this form is not limited to local usage and can be accessed as a web app via an HTTP call. This feature enhances the accessibility and versatility of the form, allowing developers to utilize it in various environments and platforms.

Form Content

The form collects the various parameters, proposed by design, from the developer. These include JMeter (threads, rampup, loops), OpenAI (GPT model, maximum token, temperature) and pipeline (pipeline directory, message caption, and chat ID) configuration parameters. Each input field is labeled for the developer's understanding.

Script Generator Form

JMeter Configuration	OpenAI Configuration	Telegram Configuration	Variables Configuration
Threads: 10	GPT Model: GPT-4	Message caption: Checkout these suggestions!	Working Directory: /
Rampup: 1	Max Token: 4000	chatid: 549538016	Service connection to azure keyvault: DevOptimizeAISubscription
Loops: 10	Temperature: 0,7		Service connection to DevOptimizeAI repository: DevOptimizeAIRepository

Submit

Copy Output

Figure 4.1: Configuration Form

Functionalities

Upon submitting the form by clicking the "Submit" button, the JavaScript function "submitForm()" is called. This function collects the values entered by the user, creates a formatted string with these values, and displays the string in the output box.

There is also a "Copy Output" button, which, when clicked, copies the content of the output box to the clipboard.

In the listing 4.1 is shown an example of output provided by the form.

```
1 - stage: JMeterTest
2   condition: always()
3   displayName: JMeter Test stage
4   jobs:
5     - template: JMeter.yml@templates
6       parameters:
7         threads: '10'
8         rampup: '1'
9         loops: '10'
10
11
12 - stage: ChatGPT
13   condition: always()
14   displayName: ChatGPT stage
15   jobs:
```

```

16 - template: ChatGPT.yml@templates
17   parameters:
18     tokens: '4000'
19     temperature: '0.7'
20     model: 'GPT4'
21     workingDir: $(workingDirectory)
22     caption: 'Checkout these suggestions!'
23     chatid: '549538016'
24     azSubscription: 'DevOptimizeAISubscription'
25
26 resources:
27   repositories:
28     - repository: templates
29       type: github
30       name: apencoreply/DevOpsEmpowerTool
31       endpoint: DevOptimizeAIRepository
32
33 variables:
34   # Working Directory
35   workingDirectory: $(System.DefaultWorkingDirectory)

```

Listing 4.1: Form output example

Script explanation

The pipeline script is divided into two stages: JMeterTest and ChatGPT.

- JMeterTest Stage: This stage is always executed and it uses a job template called 'JMeter.yml' that is stored in the 'templates' repository. The parameters for this job include:
 - the number of threads (threads: '10'),
 - the ramp-up period (rampup: '1'),
 - the number of execution loops (loops: '10').
- ChatGPT Stage: Similar to the first stage, this stage is always executed and it uses the 'ChatGPT.yml' template from the 'templates' repository. The parameters for this job include:
 - the number of tokens (tokens: '4000'),
 - the temperature setting (temperature: '0.7'),

- the model to be used (model: 'GPT4'),
- the working path, so the path in which is located the Azure Function code,
- a caption (caption: 'Checkout these suggestions!'),
- a chat ID (chatid: '549538016'),
- a azure subscription service connection string to the azure keyvault (Service connection to azure keyvault: 'DevOptimizeAISubscription').
- and a GitHub service connection string to the repository (Service connection to the repository: 'DevOptimizeAIRepository').

The resources section specifies the 'templates' repository, which is a GitHub repository named 'apencoreply/DevOpsEmpowerTool'. This repository is accessed through the 'DevOptimizeAIRepository' endpoint, configured via service endpoint connection.

The variables section defines global variables that are used in the pipeline.

- **workingDirectory:** sets the working directory for the pipeline concatenating the root directory of the pipeline with the path to reach the Azure Function code.

Here more detail about important features:

- **azSubscription:** An Azure Subscription (azSubscription) refers to a logical unit of Azure services, linked to an account and billing information, where you can deploy resources like virtual machines (VMs), databases, and more. In script, azureSubscription: 'DevOptimizeAISubscription' specifies the Azure Resource Manager connection that was created during the pipeline's creation. This allows DevOptimizeAI pipeline to interact with and manage resources in this specific Azure subscription. In particular this will be used to access the Key Vault and the GPT model.
- **endpoint:** These are repository that is accessed through a specific network endpoint. In the context of the script, the 'templates' repository, accessed through the 'DevOptimizeAIRepository' endpoint, is an example of this. This GitHub repository named 'apencoreply/DevOpsEmpowerTool' is where the script fetches job templates 'JMeter.yml' and 'ChatGPT.yml' which contain respectively the JMeter stage and the OpenAI stage. Also for the endpoint to the GitHub repository, as for the Azure Subscription, it is necessary to configure the connection during the pipeline's creation.
- **Azure Key vault:** Azure Key Vault is a service provided by Microsoft Azure for securely storing and accessing secrets. A "secret" could be anything that you want to tightly control access to. For DevOptimizeAI the secrets are endpoint and key to the GPT models.

4.2 DevOptimizeAI pipeline stages

In these sections are presented the main component of DevOptimizeAI pipeline that work to produce the test results (JMeter stage) and the response with the feedback to the Developer (OpenAI stage).

In the following subsection are presented the steps of the stages in detail and in Figure 4.2 is presented a flow chart is represented to outline the main steps in logical and temporal order.

JMeter stage

This subsection provide a description of the YAML code that is used to orchestrate the execution of the JMeter test in an Azure Pipeline. The YAML code is divided into several sections, each performing a specific role in the pipeline.

The YAML file is used to configure a job in an Azure Pipeline that installs Apache JMeter, sets parameters for JMeter at runtime, runs a JMeter test, filters and concatenates test results, and publishes the test results as a pipeline artifact.

The full code used to run the JMeter tests is in Listing A.1 in Appendix A.

Parameters

The parameters of the stage are defined at the top of the file. These parameters include threads (the number of users simulating the test), rampup (the time taken to get all threads up and running), and loops (the number of times to execute the test).

Installing Apache JMeter

The first step employs a PowerShell script to download Apache JMeter from the official archive and extract it into the working directory.

Setting Parameters at Runtime

The pipeline uses a PowerShell script to dynamically replace default values in a JMX file with parameters defined at the beginning of the YAML file. This allows for customization of the JMeter test plan based on specific requirements.

The script operates on three key parameters: the number of threads, the ramp-up period, and the number of loops. The JMX file path and the values of these parameters are obtained from the system's default working directory and the parameter values in the YAML file, respectively.

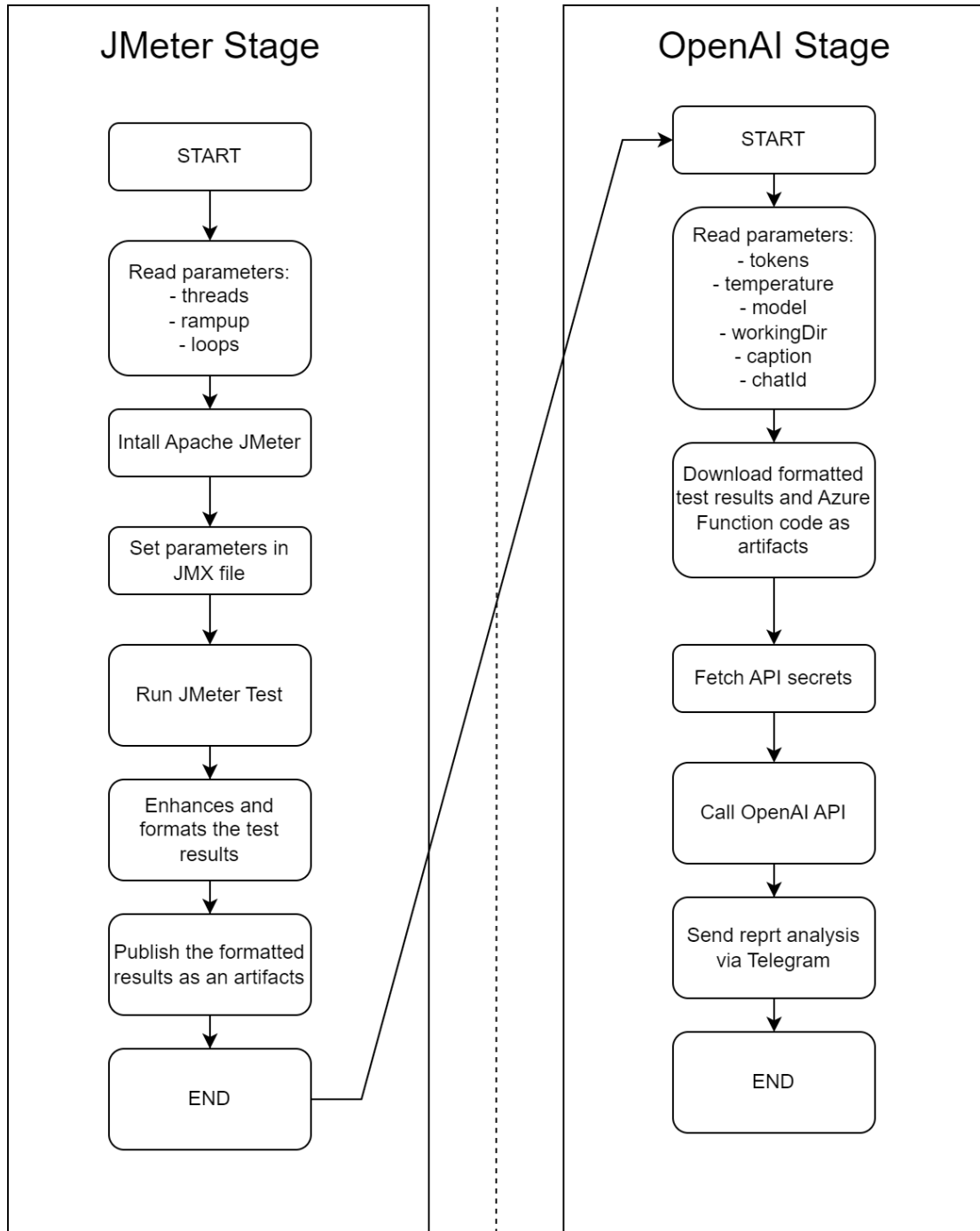


Figure 4.2: JMeter and OpenAI stages flow chart

The content of the JMX file is read and the default values for the number of threads, ramp-up period, and number of loops are replaced with the defined parameters using the -replace operation.

This dynamic modification allows the JMeter test plan to be adjusted based on the specific parameters set at runtime, providing flexibility in performance testing.

Running the JMeter Test

The pipeline proceeds with the execution of the JMeter test, facilitated by a script that uses the updated JMX file. The JMX file, which defines the test plan, is passed as an argument to the JMeter command.

The script runs in non-GUI mode (-n), which is suitable for running performance tests. The -t option specifies the JMX file to be used for the test. The JMeter test results are written to a JTL file (-l), which is a common format for recording test results in JMeter.

In addition, the script redirects the standard output to a TXT file. This file provides a summary of the test results, making it easier to quickly evaluate the performance of the tested system.

Both the JTL file and the TXT file are saved in the system's default working directory. Examples of these files can be found in Appendix D, specifically in sections D.1 and D.1 respectively.

Through this script, the pipeline is able to execute the JMeter test and capture the results in a structured and easily digestible format.

Enhances and formats Test Results

This stage employs a Python script to enhance and format summary and JTL files, removing extraneous information, and then concatenates the filtered results into a unified text file. This is a pivotal step in translating voluminous raw data into a concise and meaningful summary of the test results.

The script performs three main functions:

1. A function shifts through the summary file keeping only the relevant information.
2. A function analyse the JTL file, isolating and storing lines that encapsulate error responses.
3. Finally, a last function merge the content from both pruned files into a single, well-structured output file.

The principal advantage of this step is that it allows for a shorter and more dense version of the test results to be presented to the subsequent GPT model. The GPT model, guided by its prompt, can then generate insights more efficiently due to the reduced complexity and length of the input. This refined form of the test data not only ensures the model's performance is optimized, but it also enhances the clarity and relevance of the insights generated.

An example of this streamlined result file is provided in Appendix D, specifically in Section D.2. This sample illustrates the impact of the Python script, showcasing its ability to distill extensive and complex data into a comprehensive and easily digestible summary.

Publishing the Test Results

The final step in the pipeline publishes the enhanced test results as a pipeline artifact, allowing for subsequent download and review of the test results after the pipeline has run and for the subsequent job to retrieve these information.

OpenAI stage

This subsection provide a detailed description of the YAML code that is used to orchestrate the execution of the OpenAI API in an Azure Pipeline.

The YAML file is used to set up a job in an Azure Pipeline that interacts with the OpenAI API and sends the result to a Telegram chat.

You can find the full code used to interact with OpenAI model in Listing A.2 in Appendix A.

Parameters

The parameters are defined at the top of the file. These parameters include tokens (the maximum length of the model-generated content), temperature (controls randomness of the model's output), model (the type of model to be used), workingDir (the working directory for the pipeline), caption (the caption for the Telegram message), and chatid (the id of the Telegram chat where the message will be sent).

Downloading Artifact

The first step downloads the enhanced and formatted test result previously generated. It also need to retrieve the Azure Function Code.

Fetching Secrets

The next step fetches the API key and URL for the specified model from the Azure Key Vault.

Calling OpenAI API

The OpenAI API is interacted with through a PowerShell script. This script is designed to prepare a payload for the API request, which includes data from the enhanced test results and Azure Function code. These are read from files in the system's default working directory and the specified working directory, respectively.

The script uses the specified model's API key and URL, obtained from the Azure Key Vault, to make a POST request to the OpenAI API. The payload for this request is prepared in a specific format that includes a system message describing the role of the AI model, and a user message asking the model to provide improvement suggestions based on the formatted test results and the function code.

This prompt preparation process is vital as it directly influences the effectiveness and relevance of the GPT model's analysis. A well-prepared prompt enables the model to generate a comprehensive and tailored analysis report that provides developers with actionable insights and recommendations for optimizing their Azure Function code and improving its performance. Therefore, careful attention is paid to the crafting of the prompt to ensure the GPT model is set up to deliver the most valuable analysis.

The response from the API is then processed. The generated text is extracted and saved to a markdown file named "generatedText.md" in the system's default working directory for further use. This process allows the script to utilize the power of the OpenAI model to generate useful insights from the provided formatted test results and function code.

Sending Analysis Report via Telegram

The final stage of the process involves delivering the AI-generated insights to a specified Telegram chat. This is facilitated by a Python script that uses the requests library to interact with the Telegram Bot API.

The script is configured with the bot token, chat ID, document path, and caption as parameters. The bot token is a unique identifier provided by Telegram for the bot. The chat ID corresponds to the specific Telegram chat where the document will be sent. The document path points to the "generatedText.md" file in the system's default working directory, which contains the insights generated by the AI model. The caption is a brief message that accompanies the document in the Telegram chat.

The script constructs the API URL using the bot token, and opens the markdown file in read-binary mode. It then prepares an HTTP POST request payload that includes the chat

ID, caption, and the document itself.

The request is sent to the Telegram API. This process ensures that the AI-generated insights are delivered efficiently and securely to the specified Telegram chat, allowing for easy access and review.

The use of Telegram for delivering these insights has several advantages over traditional methods like email. Messages on Telegram are delivered in real-time and the app's push notifications ensure they are seen promptly. The platform also allows for easy group collaboration, and its robust API makes automating the delivery of insights straightforward. Furthermore, Telegram is known for its strong security measures, making it a safe option for sharing potential sensitive information like AI-generated insights.

4.3 GPT response formatting

In this section, the different potential responses from the model are discussed. Basically, the responses that the model can provide are of two types.

Please note that for brevity and clarity, the full function code has been omitted in this section. However, it is in Appendix B, specifically in Listing B.3. This code provides a deeper understanding of how the model processes the input and generates its responses.

Response without code example

The first type of response doesn't include a code example. In this case, the GPT model provides a text-based explanation, suggestion, or answer depending on the query. This could be a response to a theoretical question, an explanation of a concept, or a suggestion for improving code without providing a specific code snippet. The response is typically in a conversational format, and its main objective is to provide insights, knowledge, or guidance based on the user's input.

Here in this paragraph is shown an example of response with only plain text in it.

Based on the code snippet and JTL results, here are some possible improvements:

1. Move configurations to a separate configuration file or use environment variables: Instead of hardcoding the configuration values (`cosmosDbEndpoint`, `cosmosDbKey`, `databaseId`, `containerId`), it is better to store them in a separate configuration file or use environment variables for better flexibility and security.

2. Implement error handling and logging: Currently, the code only logs the exception when an error occurs while inserting data into Cosmos DB. It would be more useful to have proper error handling and logging throughout the code to provide better debug information and handle exceptions gracefully.

...

Implementing these improvements will help enhance the code's performance, security, and maintainability.

In the following is presented the input text that is submitted to the GPT-3.5-turbo model without system message configuration.

Result from the JTL file from the JMeter test in which are saved the errors and the description:

JTL Results:

Summary results:

summary = 100 in 00:00:12 = 8.1/s Avg: 1068 Min: 308 Max: 8042 Err: 0 (0.00%)

Function Code:

#FUNCTIONCODE

Response with code example

The second type of response includes a code example. In this case, besides providing a text-based response, the GPT model also generates a relevant code snippet. This is particularly useful in instances where the user is looking for a practical solution or wants to understand a concept through an example. The model generates code based on the context of the query and the language specified. This allows users to not only understand the concept but also see its practical implementation, enhancing their learning experience.

Here in this paragraph is presented an example of response with code suggestion in it.

Based on the given code snippet and JTL results, here are a few suggestions for improving the code:

1. Use Connection Pooling for HttpClient:

- The current code creates a new instance of HttpClient for each request, which can lead to performance issues. It is better to use a single instance of HttpClient and reuse it across requests. You can achieve this by using connection pooling.
- Modify the code to create a static instance of HttpClient with connection pooling enabled:

```
1 private static readonly HttpClient httpClient = new
  ↪ HttpClient(new HttpClientHandler()
2 {
3     UseDefaultCredentials = true,
4     MaxConnectionsPerServer = 100,
5 });
```

2. Optimize Cosmos DB Client usage:

- The code currently creates a new instance of CosmosClient and Container for each request. Instead, initialize these objects once and reuse them across requests.
- Modify the code to create static instances of CosmosClient and Container:

```
1 private static readonly CosmosClient cosmosClient =
  ↪ new CosmosClient(cosmosDbEndpoint, cosmosDbKey
  ↪ );
2 private static readonly Microsoft.Azure.Cosmos.
  ↪ Container cosmosContainer = cosmosClient.
  ↪ GetContainer(databaseId, containerId);
```

...

These are some general suggestions to improve the code based on the provided information. It is important to consider the specific requirements and constraints of your application while implementing these suggestions.

In the following is presented the input text that is submitted to the GPT-4 model with system message configuration.

Provide improved implementation example taking into account the following:

Result from the JTL file from the JMeter test in which are saved the errors and the description:

JTL Results:

Summary results:

summary = 100 in 00:00:12 = 8.1/s Avg: 1068 Min: 308 Max: 8042 Err: 0 (0.00%)

Function Code:

#FUNCTIONCODE

Chapter 5

Experimental Results

This chapter will delve in different stages of testing.

It starts by examining the Azure Functions used to test DevOptimizeAI in Section 5.1. This will provide a clear understanding of how these functions contribute to the robust testing of the system, helping to ensure its reliability and efficiency.

Then the Section 5.2 provide information about the acceptance testing carried out to guarantee that the requirements are met. Meanwhile the section explores different test configurations for both JMeter tests parameters configuration and model type.

Subsequently, in Section 5.3, the focus will be on validation testing for the efficiency and improvement of the AI component of DevOptimizeAI.

Specifically, all with the aim of understanding how to get the best response, the following tests will be carried out:

- Firstly, **prompt selection/engineering tests**, to try to explore the best prompt message for the GPT Models.
- Then, **CPU-Intensive test**, to understand the efficacy of DevOptimizeAI for this type of API.
- Subsequently, **behavioural tests**, to try to exploit the best from the model using different approach like a reuse of the feedback provided or give a 'hint' in the code to the GPT.
- And finally a **comparison tests**, with the direct competitor: ChatGPT.

In Table 5.1 are presented an overview with all the category of test that will be conduct during this chapter with a brief description and the main outcome.

Test Name	Detailed Description	Comprehensive Key Insights
Acceptance Test	This test is designed to ensure that DevOptimizeAI meets the predefined requirements. It involves exploring different configurations and checking the system's response to ensure compliance with the requirements.	The test confirmed that DevOptimizeAI adheres to the specified requirements under various configurations.
Prompt Testing	This test examines how DevOptimizeAI responds to a variety of prompts and measures the influence of these prompts on the system's performance. It helps in understanding the versatility and adaptability of the system.	The test revealed that DevOptimizeAI responds effectively to different prompts, demonstrating high flexibility and adaptability in its operations.
CPU-intensive Test	This test is focused on evaluating the performance of a CPU-intensive function in DevOptimizeAI. It includes injecting logical errors and inducing eternal loops to understand how effectively DevOptimizeAI can analyze and handle such scenarios.	The results indicated that CPU-intensive functions are more beneficial when using DevOptimizeAI as it effectively identifies and handles injected errors and infinite loops.
Behavior Testing	This test evaluates how DevOptimizeAI behaves under a range of conditions. These include varying developer abilities, code cues, successful answer examples, and repeated requests. It gives insights into the system's robustness and resilience.	The test showed that DevOptimizeAI performs well under most conditions. However, it underperforms in a few scenarios.
ChatGPT vs DevOptimizeAI	This test compares the performance of DevOptimizeAI against ChatGPT, a widely recognized language model developed by OpenAI. It provides insights into the comparative effectiveness of the two systems.	The comparative study offered valuable insights into the relative strengths and weaknesses of both systems.

Table 5.1: Comprehensive Overview of Tests Conducted

5.1 Azure Functions Test Code Implementation

This section discusses how Azure Functions are implemented and tested within the DevOptimizeAI. For the purpose of testing DevOptimizeAI, the focus moves on the HTTP-triggered Azure Functions because Apache JMeter is designed to load test web APIs.

To assess the capabilities of the DevOptimizeAI, GPT Model and JMeter, have been created and tested a variety of HTTP-triggered Azure Functions. These functions simulate various real-world scenarios that a typical web API might encounter. This approach ensures comprehensive testing of DevOptimizeAI pipeline.

Test Azure Functions

- **GetAPI:** This function responds to an HTTP request with a simple message. It's useful for testing the basic functionality of the pipeline and JMeter's ability to send requests and receive responses.
- **PostAPI:** This function takes an HTTP request, processes data contained in the request body, and returns a response. This can simulate more complex operations that a web API might perform.
- **GetMean:** This function accepts an HTTP GET request with a specific parameter. It retrieves related data, performs an analysis on the data, and returns the result in the HTTP response. This scenario represents complex operations involving both data retrieval and computation.

These scenarios represent typical API scenarios, but are predominantly I/O-intensive, not requiring a large CPU load to execute, but exchanging a large amount of data. For this reason, tests were also conducted on CPU-intensive functions.

- **CalculatePrimeNumbers:** This function performs a computation-intensive task upon receiving an HTTP request. It could involve complex mathematical calculations or other tasks that require significant CPU resources. The result of the computation is returned in the HTTP response.

Azure Functions test plans

The next step is to configure JMeter test plan (JMX file) to JMeter stage to send HTTP requests to these functions and capture the responses.

For each function, they have to assess the following:

- **Response Time:** How long does it take for the function to respond to the request? This is a key performance indicator.
- **Success Rate:** Does the function always respond successfully under load, or are there any failed requests?
- **Throughput:** How many requests can the function handle per unit of time?

Detailed Implementation of Azure Functions

The subsequent paragraphs delve into the details of each API, explaining their operations and their contribution to the comprehensive testing of the DevOptimizeAI prototype.

Upon initialization, all these three I/O intensive functions create a CosmosClient and Cosmos Container using the Cosmos DB endpoint, key, database ID, and container ID obtained from the environment variables.

After this common initialisation, the functions differ as follows.

GetAPI

The *GetAPI* function (Appendix B.2) is a data retrieval mechanism that interfaces with the Azure Cosmos DB service to obtain specific data entities based on a provided ID. The function is designed to be invoked through an HTTP GET request.

In Figure 5.1 is presented a Flow Chart to visually present this API.

The function accepts an HTTP request object and extracts the 'id' parameter from it. A validation check is performed to ensure that the 'id' is provided, thereby ensuring the integrity of the function execution.

The function then attempts to retrieve the item from the Cosmos DB using the provided 'id'. If successful, the retrieved item is returned in the HTTP response as an *OkObjectResult*.

In situations where the provided 'id' does not correspond to any item in the Cosmos DB, the function captures the CosmosException and returns a *NotFoundResult*. For all other exceptions, an error message is logged, and an HTTP 500 status code is returned, indicating an internal server error.

The *RequestDocument* and *Tag* classes encapsulate the schema of the data stored in the Cosmos DB, facilitating structured data retrieval and deserialization.

The *GetAPI* function tests the scenario of HTTP GET requests and data retrieval operations from Azure Cosmos DB; it also satisfies the Simple HTTP Trigger scenario. It responds to

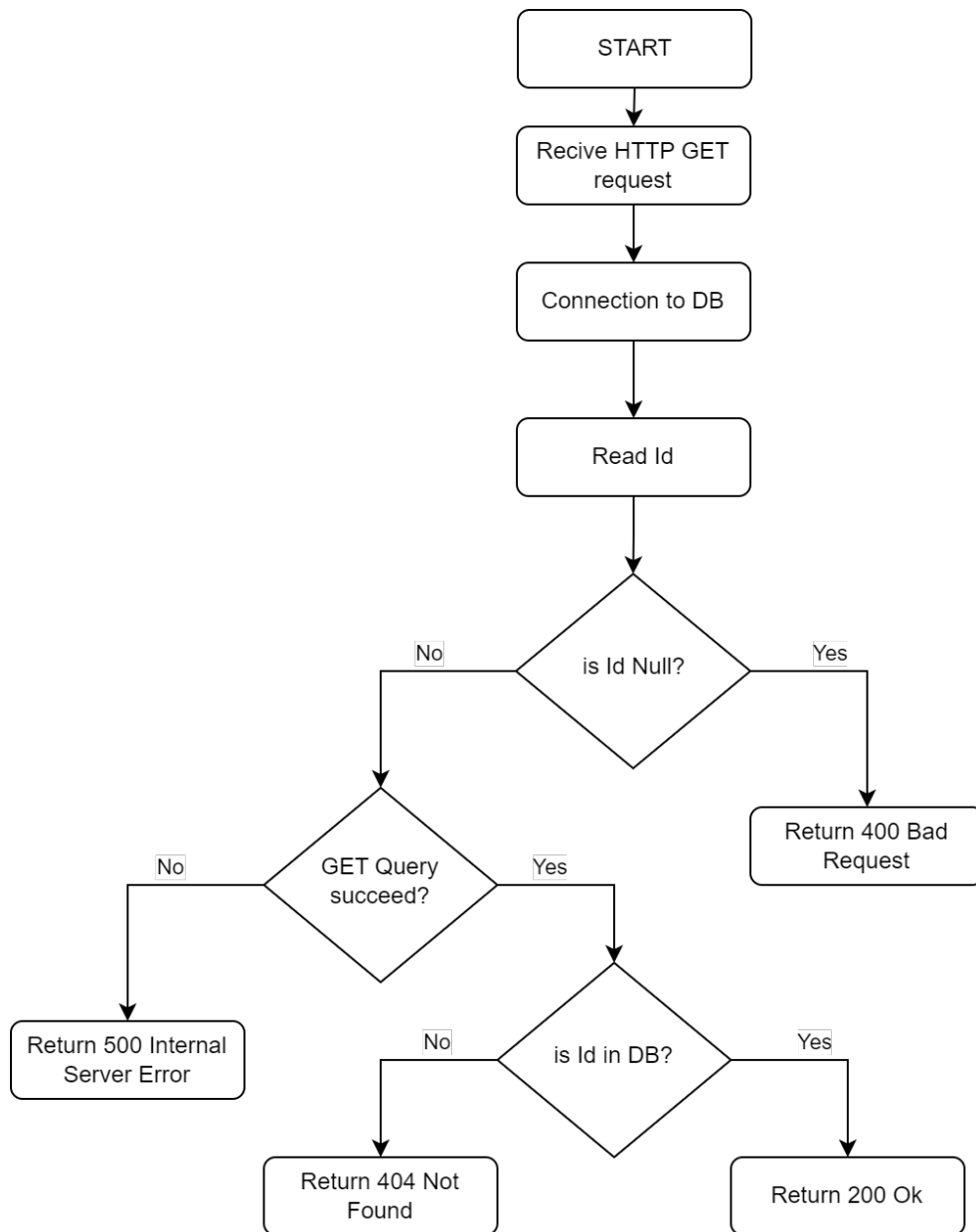


Figure 5.1: GetAPI - API Flow Chart

an HTTP GET request and retrieves data from Azure Cosmos DB. However, it does more than just returning a simple message; it's involved in database operations.

The JMX file outlines a test plan that will execute a GET request to:

- `getwebapi.azurewebsites.net/api/GetWebAPI` (the API endpoint)

with variable parameters such as the number of threads, the ramp-up period, and the loop count. The test plan will generate a random number called "ID" between 100 and 1000 for each HTTP request. The results are collected and displayed in a Summary Report.

For the complete description of the JMeter test for this function, please refer to the JMX file in Listing C.2 in Appendix C.

PostAPI

The *PostAPI* function (Appendix B.3) serves as a mechanism for data insertion, designed to interface with the Azure Cosmos DB service. The function is invoked by an HTTP POST request and is responsible for inserting a new data entity into the Cosmos DB.

In Figure 5.2 is presented a Flow Chart to visually present this API.

The function accepts an HTTP request object and reads the JSON data from the request body. This JSON data is deserialized into a RequestDocument object.

Before data insertion, the function validates that the 'id' attribute of the RequestDocument object is provided, ensuring data integrity.

The function then attempts to insert the RequestDocument object into the Cosmos DB. A successful insertion results in the function returning an *OkObjectResult*, signifying the successful data insertion.

In the event of exceptions during execution, the function logs an error message and returns an HTTP 500 status code, indicating an internal server error.

The *PostAPI* function tests the scenario of HTTP POST requests and data insertion operations into Azure Cosmos DB; it also satisfies Data Processing scenario. It receives an HTTP POST request, processes data contained in the request body (JSON data), and inserts it into Azure Cosmos DB, returning an HTTP response.

The JMX file for the *PostAPI* outlines a test plan that will execute a POST request to:

- `fidelapp.azurewebsites.net/api/FiDelApp` (the API endpoint).

This test plan also includes variable parameters such as the number of threads, the ramp-up period, and the loop count. This test plan differs from the one for GetAPI in that it

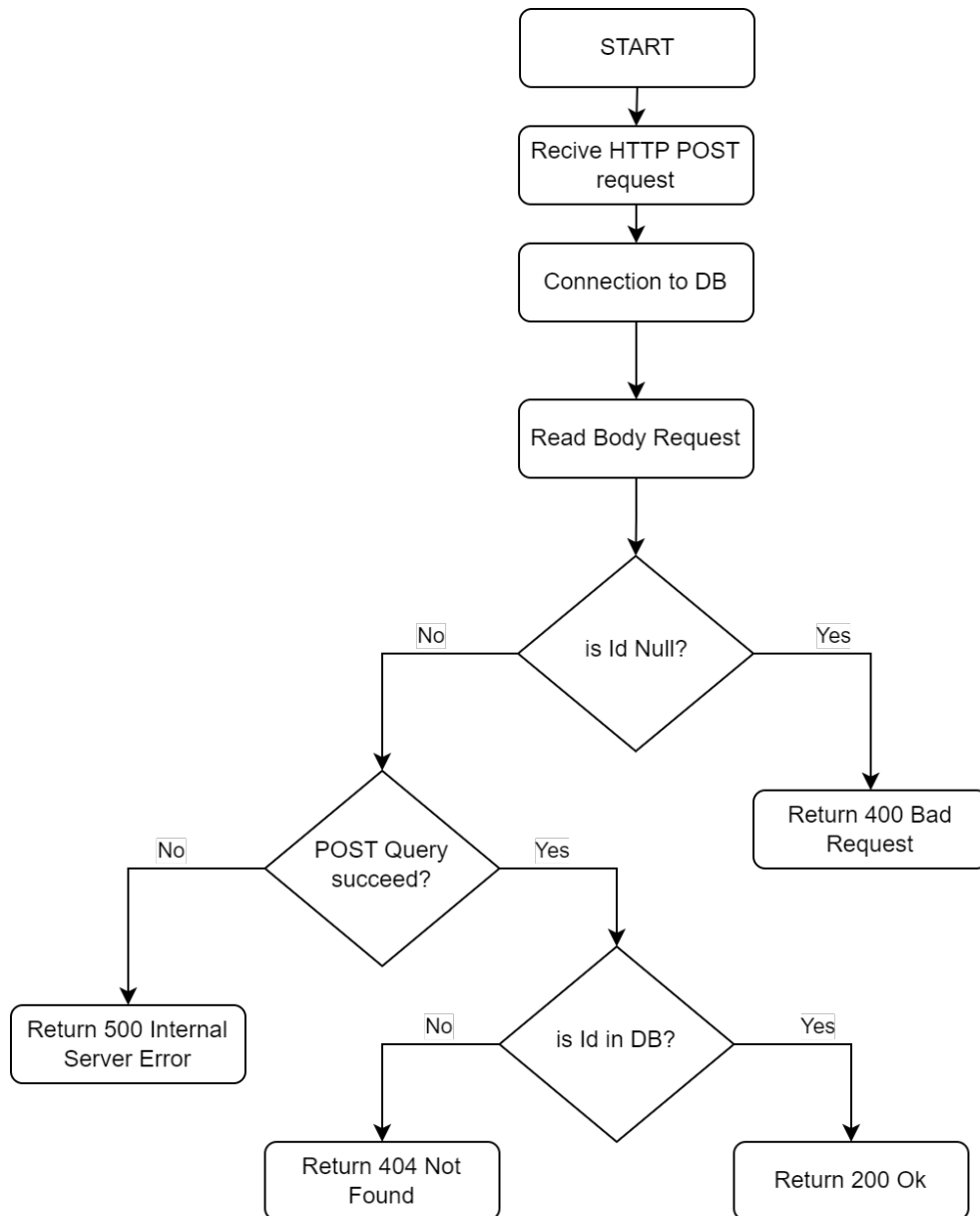


Figure 5.2: PostAPI - API Flow Chart

includes a payload with each HTTP request. The payload is a JSON object which includes a variable that represent the 'id' of the Cosmos DB that is managed via GUID.

The corresponding JMeter test is detailed in the JMX file in Listing C.3 in Appendix C.

GetMean

The *GetMean* function (Appendix B.1) is an Azure Function designed to compute the mean elevation of volcanoes for a specified country. The function operates by interacting with the Azure Cosmos DB service, extracting the necessary data, and performing the statistical calculation.

In Figure 5.3 is presented a Flow Chart to visually present this API.

Upon receiving an HTTP GET request, the function extracts the 'Country' parameter. It validates the 'Country' parameter to ensure it is not null or empty.

A FeedIterator of type Volcano is then initialized to traverse the Cosmos DB records. The function iterates over each record, checking if the 'Country' attribute matches the provided parameter. If a match is found, the function attempts to parse the elevation to an integer, incrementing the total elevation sum and the count of volcanoes.

Upon completion of the iteration, the function computes the mean elevation by dividing the total elevation sum by the count. The mean elevation is then returned as an *OkObjectResult*. In the event of any exceptions during the execution, the function logs an error message and returns an HTTP 500 status code, indicating an internal server error.

The *GetMean* function satisfies Data Analysis scenario. It receives an HTTP GET request with a specific parameter, retrieves the relevant data from the database, performs the mean value, and returns this result in the HTTP response. This process embodies the operations of data retrieval, analysis, and response generation.

The JMX file outlines a test plan that will execute a GET request to:

- `getmean.azurewebsites.net/api/GetMean` (the API endpoint)

with variable parameters such as the number of threads, the ramp-up period, and the loop count. This test plan differs from the other two in that it includes a variable "Country" parameter with each HTTP request to determine a sample country to the test call. The results are collected and displayed in a View Results Tree.

The JMeter test that targets this function can be found in Listing C.1 in Appendix C.

CalculatePrimeNumbers

The *CalculatePrimeNumbers* function (Appendix B.4) is a CPU-intensive Azure Function

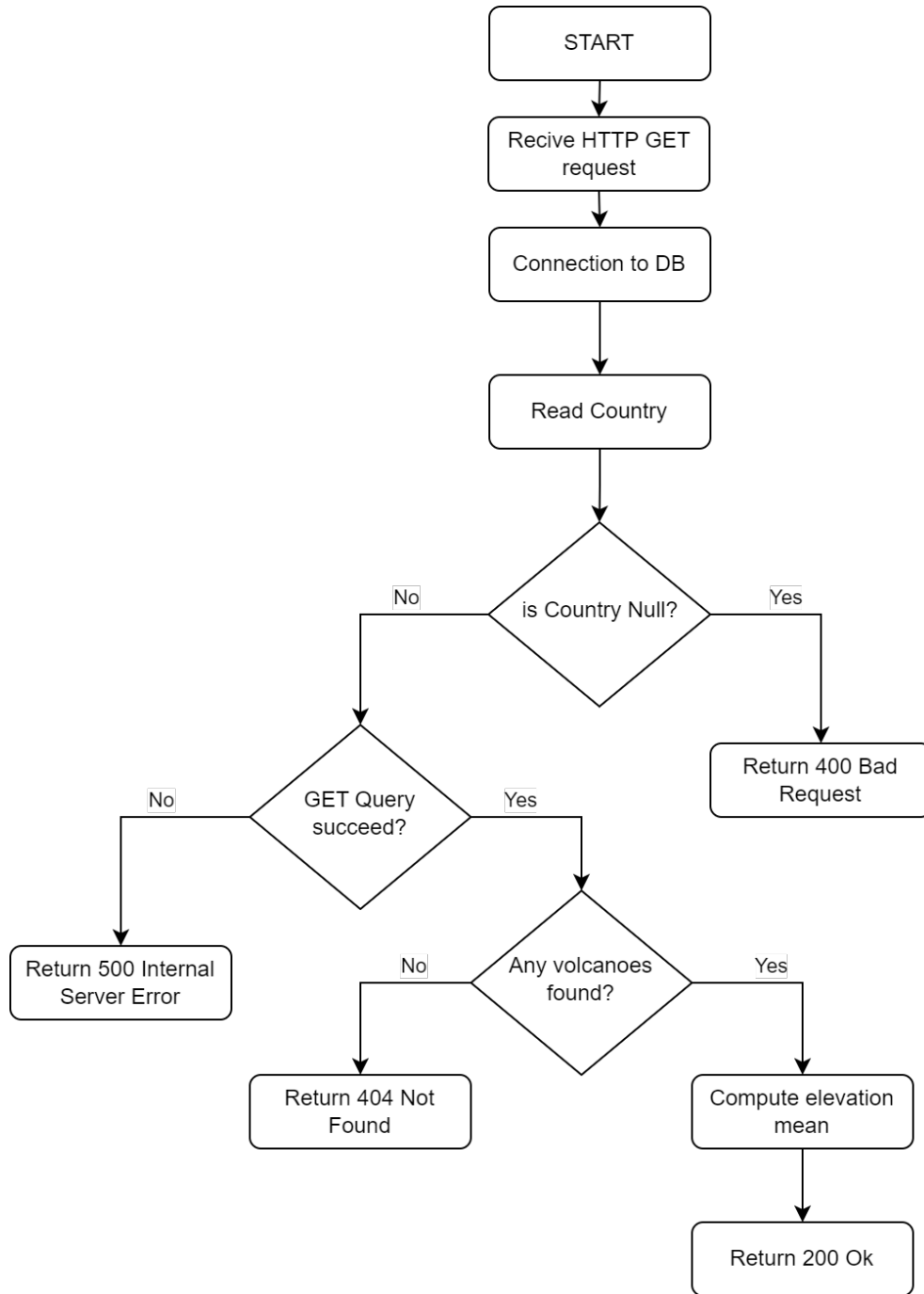


Figure 5.3: GetMean - API Flow Chart

designed to compute prime numbers up to a specified number. This function is designed to stress the CPU resources and test the system's performance under computation-intensive tasks.

In Figure 5.4 is presented a Flow Chart to visually present this API.

Upon receiving an HTTP GET request, the function extracts the 'number' parameter from either the query string or the request body. The parameter is validated to ensure it is a valid integer.

The function then calls the `GetPrimes` function, passing the validated 'number' as the argument. The `GetPrimes` function uses the Sieve of Eratosthenes algorithm to find all prime numbers up to the specified number. It initializes a list of prime numbers with the first prime number, 2. For each odd number up to the 'number', it checks whether the number is divisible by any of the known primes. If a number is not divisible by any known prime, it is prime, and is added to the list of primes.

Once all the primes up to the specified number have been computed, the function returns these prime numbers in an HTTP response as an *OkObjectResult*.

In case the 'number' parameter is not provided or is not a valid integer, the function returns a *BadRequestObjectResult*, indicating that a valid integer should be provided.

The *CalculatePrimeNumbers* function tests the scenario of CPU-intensive tasks. It receives an HTTP GET request with a specific parameter and performs a computation-intensive task, returning the result in the HTTP response.

The JMX file outlines a test plan that will execute a GET request to:

- validationtestfunctions.azurewebsites.net/api/CalculatePrimeNumbers (the API endpoint)

with variable parameters such as the number of threads, the ramp-up period, and the loop count. This test plan includes a variable "number" parameter with each HTTP request to determine the upper limit for the prime numbers calculation. The results are collected and displayed in a View Results Tree.

The corresponding JMeter test is detailed in the JMX file in Listing C.4 in Appendix C.

5.2 Acceptance tests

To conduct acceptance tests, it is necessary to try DevOptimizeAI with all real world scenario test cases, so the I/O-intensive functions and different configurations. The CPU-intensive function will be tested in the validation test part in another manner. To do this,

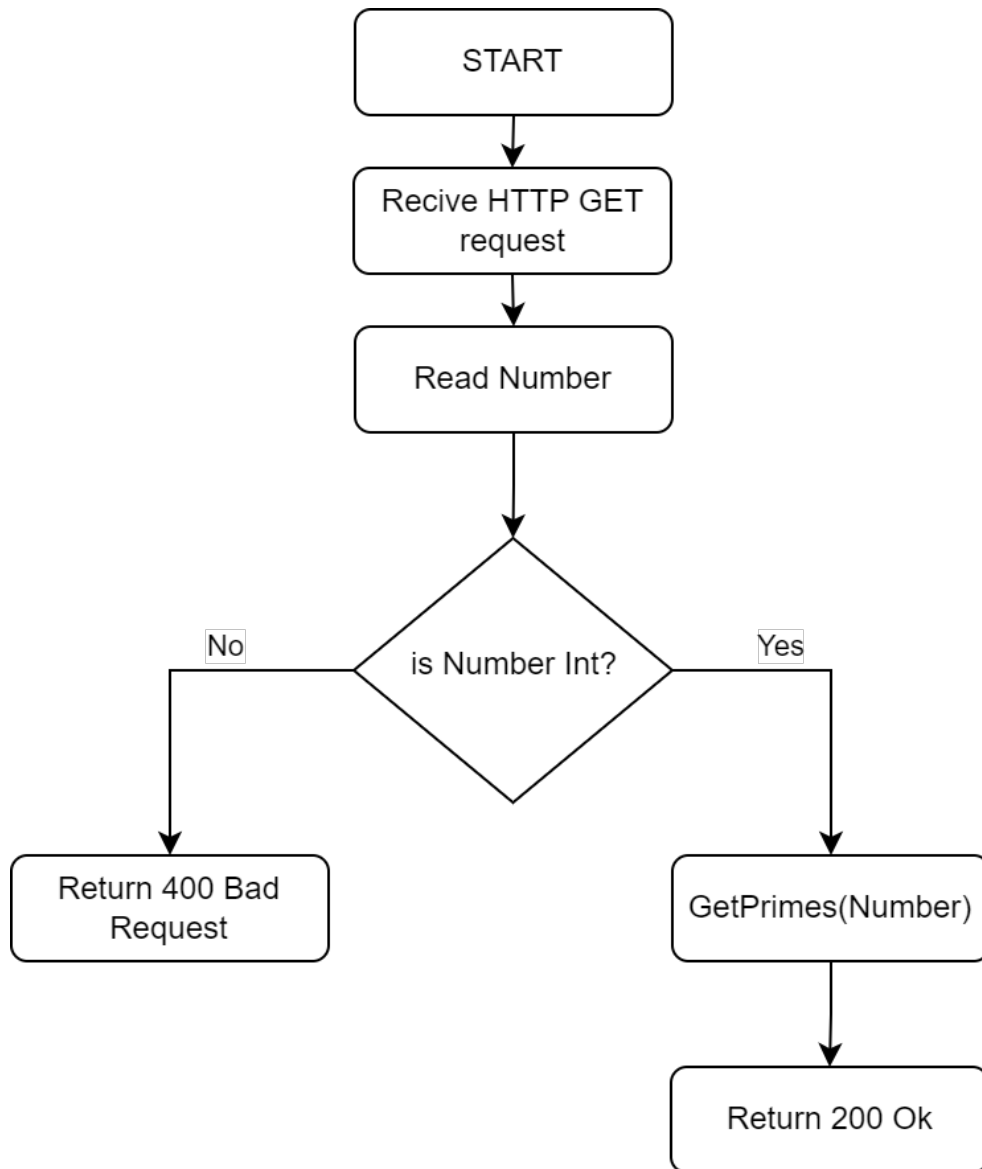


Figure 5.4: CalculatePrimeNumbers - API Flow Chart

an Azure DevOps pipeline was developed for each Azure Function code with associated test plans, and in each pipeline a connection was set to the DevOptimizeAI pipeline. Presented below are the configurations and results for the tests regarding the variation of:

- Azure Function code
- GPT Model
- Max token response
- Different number of threads and loops.

For each test is it possible to consult the Appendix E to read all the response provided by DevOptimizeAI.

An example of the input to the model is presented in Chapter 4 in Section 4.3 when *Response without code example* is discussed.

GPT-3 tests

DevOptimizeAI is tested using GPT-3 model with the three previously mentioned functions: *GetMean* B.1, *GetAPI* B.2 and *PostAPI* B.3.

The purpose of this test is to understand how DevOptimizeAI work with different environmental request; in this case the aim is to compare two GET function, one more complex and another very simple, and a POST and how the GPT-3 model perform this requests.

The result of this test can be seen in the Appendix E in the Section E.1.

For the execution of this test has been used the GPT-3 model with 4000 token available to try DevOptimizeAI whit low usage of resources, it is also not done any type of specific request, to the model are only submitted the test results and the code. The response that DevOptimizeAI provide are quick, but not specific for the purpose of the enhancing of the API. The response only provide a brief description of the provided texts, without any snippet of code proposed by the model itself.

A summary of quantitative result obtained from these test are presented in Table 5.2.

GPT-4 tests

Immediately after testing DevOptimizeAI with the GPT-3 model it is tested DevOptimizeAI with the GPT-4 model.

The answers obtained are in the Appendix E in the Section E.2.

Several factors can be seen in these responses that define that these responses are better than previous responses.

First of all a greater size of the response text: which is also richer in information and with a better formatting of the text that makes it more understandable.

It also provides a more detailed description of the function in question going, although in a generic way, to provide advice not related with performance only, but also on possible checks and changes to be made to the code to enhance it, also based on the results of the tests.

A summary of quantitative result obtained from these test are presented in Table 5.2.

Model	API	AVG Re- sponse Time x call (ms)	Scalability (Requests x second)	JMeter test time (sec)	Response Size (KB)
GPT-3	GetMean	3267	2.7	37	0.9
GPT-4		3442	2.6	39	1.2
GPT-3	GetAPI	890	7.3	14	0.8
GPT-4		448	10.8	9	1.1
GPT-3	PostAPI	923	7.7	13	0.7
GPT-4		400	13.2	8	1.3

Table 5.2: Comparison of GPT-3 and GPT-4 Tests with 4k max token response

Response token test with GPT-4

After testing the GPT-4 model the following tests will focus on exploiting all its potential going to see how far you can push the model without wasting resources or running into errors shifting the focus from model selection to the selection of other parameters such as the Max Token for the GPT Model or the values of threads and rampup for the JMeter test.

A first test was carried out by modifying the maximum number of tokens available for the response and thus allowing the model to provide more extensive answers. The maximum amount of token include the token in input (the user message) and output (the response) of the model. These tests are conducted using GPT-4 model with available 32 thousands of token.

The model having 32k token if the response token are set to 14k, that means that for the input tokens the model could provide at most 8k tokens.

These tests have not led to satisfactory results although having greatly increased the number

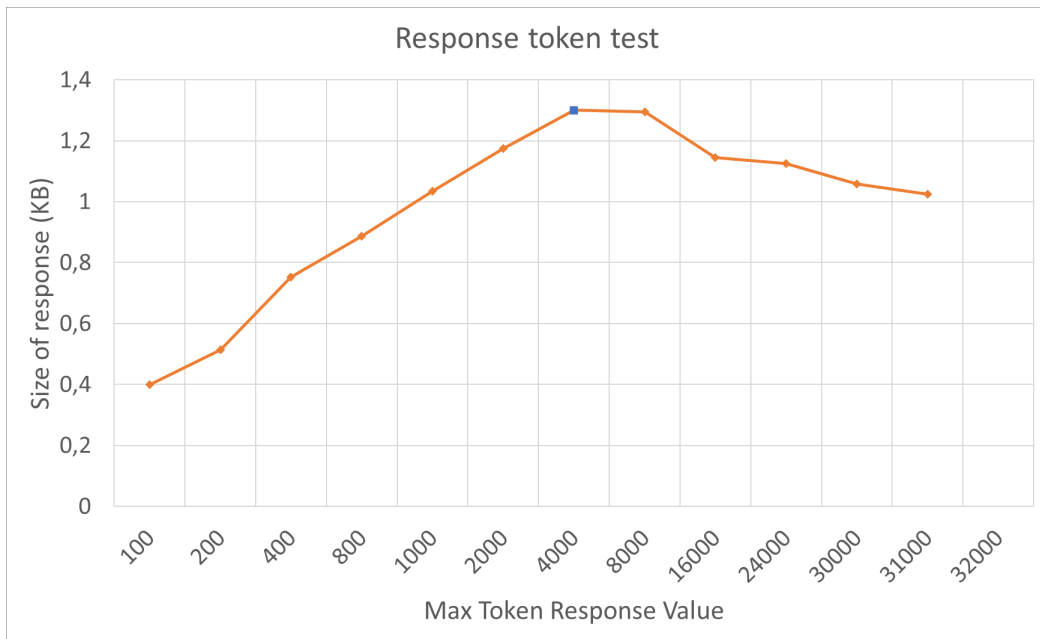


Figure 5.5: GPT response dimension varying the max token value

of maximum tokens available for the response the quality of the response has remained almost the same both in terms of length and in terms of clarity and effectiveness to the developer, also for the similarity with the previous response, these response are not present in the appendix.

A representation of what just said is shown in the Figure 5.5 in which are condensate several test on the three different functions.

In the figure it is shown how the response size grow rapidly until 4000 tokens for the response and an after that stay between 1 and 1.2 KB of message response, with a slightly decreasing approaching the model limit and going in error if passed.

The Table 5.3 give another view of the same data.

While a higher token limit allows the model to generate longer responses, it doesn't necessarily mean that the model will have a better understanding of the input or be able to provide more accurate or relevant responses. The model's performance is based on its ability to understand and generate contextually appropriate responses, not the length of the response.

Lower amount of token can cause a truncate response. It is the case when a value lower than 1k token is submitted to DevOptimizeAI.

This parameter is more usefull with longer functions, these will benefits the most form

Max Token Response Value	Weight (KB)
100	0.4
200	0.5
400	0.7
800	0.8
1000	1.034
2000	1.175
4000	1.300
8000	1.294
16000	1.144
24000	1.125
30000	1.058
31000	1.024
32000	NAN

Table 5.3: Response token test table

the decreasing of this parameter, because allow also to them to obtain a response without leading the model into error, even if eventually shorter compared to some shorter function that use higher value for this parameter.

Load test GPT-4 test

Following the tests conducted with the GPT-4 model and varying maximum token amount, the next phase of testing involved increasing the number of API calls. The goal was to understand the model's performance under a higher load and determine its capability in handling a larger volume of requests.

In these tests, the GPT-4 model was subjected to a higher number of API calls, while maintaining the other parameters constant: the model is GPT-4 and the max token response are 8000. The model was expected to process these calls and provide responses within the defined token limit. The quality, clarity, and efficiency of these responses were then evaluated, similar to the previous tests.

Each result is the mean of the three different executions, one for each function, providing a more robust analysis of the system's performance, and also normalize to make all the parameter comparable.

As part of this testing phase, a number of cases were designed where the number of API calls was increased. To visualize the model's performance in these cases the Figure 5.6 shows result obtained clustered by the number of API calls as threadsXloops.

So if in the plot is presented 100x1000 that means that the test is conducted with 100

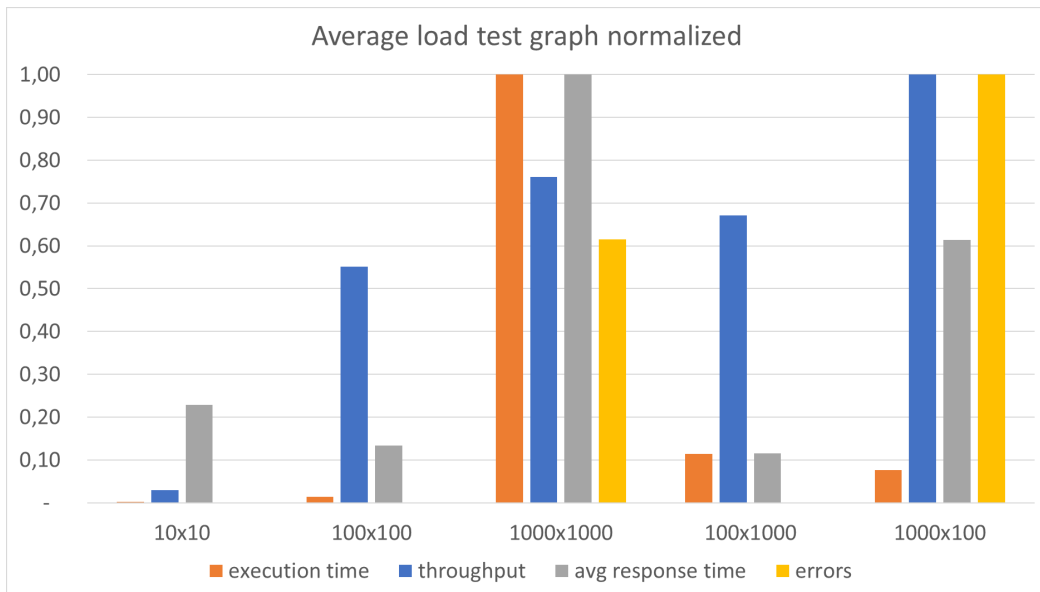


Figure 5.6: Load Test Graph

threads executing 1 API call for 1000 times. Rampup is setted to 1.

This graph provides a clear visual representation of how the model's response times are affected as the number of API calls increases.

The 10x10 and 100x100 tests had the shortest execution times and zero errors, indicating excellent performance and reliability when handling smaller test scenario.

The 100x1000 test showed impressive results with a high throughput of 269.6 messages per second and the shortest average response time of 363 milliseconds, all with no errors. This suggests that the applications excel at handling a large volume distributed over different calls round.

However, the 1000x1000 and 1000x100 tests showed significant increases in execution time and error rates, suggesting challenges when dealing with high volumes of threads regardless of loops. The 1000x100 test had the highest throughput at 401.9 messages per second, but also the highest error rate at 69.53%.

In addition to the graph, Table 5.4 shows the key data from these tests. This table includes information such as the number of API calls, the response times, and other relevant metrics. It provides a quick and easy way to compare the results across different test cases and identify any trends or patterns.

However, it should be noted that the response messages from these tests are not included in the Appendix E. This is because the responses are analogous to those from the previous

Test	Time (sec)	Throuput (msg x sec)	avg response time (ms)	errors (%)
10x10	8	12.1	721	0
100x100	45	221.6	419	0
1000x1000	3269	305.9	3146	42.75
100x1000	371	269.6	363	0
1000x100	249	401.9	1931	69.53

Table 5.4: Summary of load test results

tests, and no further prompt engineering has been done at this stage. Indeed, the focus of this test is more on the model's performance metrics under increased API calls, rather than the content of the responses.

Upon analysis of the graph and table, it is evident that as the number of API calls increases, there is a corresponding increase in the response times. However, the model still manages to maintain a relatively stable performance, indicating its robustness and ability to handle a higher load.

Test with logical error in the test plan

It has been seen that with regard to logical errors within the code of the Azure Function DevOptimizeAI is able to detect such errors and suggest a solution approach as well as an improvement approach. Now its behavior in case the logical error is present in the JMX file will be analyzed.

A test was done with the PostAPI test plan, where at the post of entering an id of type string, it was tried to enter an id of type integer which causes a write error to the DB classified as 500 Internal Error.

Trying to run DevOptimizeAI with this test plan showed that the suggestions are still related to the code, proposing solutions that improve the program, but having less specificity given by the distorted result of the JMeter test.

So despite the test plan being poorly set up, the system still manages to provide an answer to the developer, but not as targeted as if the test had been set up correctly.

Some consideration

The Table 5.5 provide an overview of the test and result done in this section. Throughout the testing process, several considerations have been drawn.

1. The model's capabilities are dependent on the number of tokens available for it to provide responses. However, increasing the maximum number of tokens does not

Test	Model	Token Limit	Comments
GPT-3 Test	GPT-3	4000	General responses, no code snippets
GPT-4 Test	GPT-4	4000	Richer responses, more detailed information
Response token test	GPT-4	N/A	The improvement in the responses with larger size are achievable, but not mandatory. Larger response size do not necessary outcome into better response.
Load test	GPT-4	8000	DevOptimizeAI demonstrated robustness and ability to handle a higher load, with a relatively stable performance despite the increase in API calls.
Logical error in test plan	GPT-4	8000	DevOptimizeAI is still able to provide suggestion to improve the API, but do not grasp any error in the test plan configuration.

Table 5.5: Summary table of integration tests

necessarily lead to more satisfactory results in terms of response quality.

2. The model's performance can also be affected by the number of requests it handles. While it is capable of handling a high number of requests, errors may occur when the number becomes too large. These errors are not necessarily due to the model's inability to handle the load, but could be attributed to issues in the code or database configuration.
3. Despite the issues encountered, the model exhibits a remarkable ability to self-diagnose and recommend improvements to the code to manage and prevent errors. This shows that the model can provide valuable insights for developers even when not explicitly asked to do so. However, it's important to note that in none of the responses did the model provide suggested code.
4. Lastly, the model's expressiveness seems to decrease when the number of input tokens increases. This suggests that the model might be more effective when used for simpler functions rather than complex ones or that very large test could not lead to more expressive analysis.

5.3 Validation tests

Now the research will delve into some validation test done with DevOptimizeAI and the three previously mentioned functions: *GetMean* B.1, *GetAPI* B.2 and *PostAPI* B.3.

DevOptimizeAI will be tested using GPT-4 with a maximum response token fixed to 14000 and the JMeter configuration fixed.

The purpose of these tests is to understand the real capabilities and limitation of DevOptimizeAI.

Prompt test

This section will explore how it is possible to enhance the input provided to the GPT model by DevOptimizeAI, for example with some prompt engineering and other test, and analyze how its response to the users is influenced.

The test will be presented each one divided by paragraph and for each test is it possible to consult the Appendix F to read all the response provided by DevOptimizeAI.

An example of the input to the model is presented in Chapter 4 in Section 4.3 when *Response without code example* is discussed.

Another example of input file to the model, this with enhanced user input, is provided in Appendix D.3.

Prompt engineering

This paragraph delves into the testing procedures undertaken to evaluate the efficiency and functionality of prompt engineering within DevOptimizeAI, specifically focusing on how the GPT-4 model interacts with different types of inputs.

To understand the influence of different input types on the model's output, was been conducted a test in which the GPT-4 model was set to interact in three distinct ways:

- Interaction with only the system message: This setup aimed to observe how the model responds when it is given only a system message as input. The system message typically provides high-level instructions or context to the model.
- Interaction with only the user message: In this case, the model was tested with only a user message as input. The user message is often more specific and task-oriented than the system message.
- Interaction with both system and user messages: This test was conducted to see how

the model handles a combination of system and user messages, to exploit the best response possible with richer information and more suited advice.

The part of the user message containing the test information and the Azure Function code remains in any test configuration. The objective of this testing procedure was to discern whether the model's output is more influenced by the system-provided context (system message), the user's specific request (user message), or a combination of both. This understanding would help to determine the optimal configuration for generating complete and consistent responses to user.

System Message prompt engineering

In this analysis, we explore the results of tests performed using the prompt engineering, applied at the system message level.

Detailed responses derived from these tests can be found in Appendix F in Section F.1.

The test on *GetMean* offered valuable insights into areas requiring enhancement, such as database management, parallel processing, and the implementation of caching, tracing, and sharding strategies, all crucial for reducing response times.

The test on *GetAPI* emphasized the importance of robust error handling and data validation in minimizing HTTP 404 errors. It also highlighted the need for a more comprehensive logging system that provides specific details about the error, such as the unique ID that was not found.

The test on *PostAPI* underscored the necessity for sturdy exception handling mechanisms to prevent HTTP 500 errors, drawing attention to the importance of logging meaningful error messages and implementing advanced error handling strategies.

It is important to note that while these tests provide valuable insights and recommendations, none of them offer specific code snippet examples. They focus on providing a high-level understanding of the issues and potential solutions, leaving the implementation details to the developers.

User Message prompt engineering

This paragraph will look at the results of the tests performed with prompt engineering at the user message level. These responses are particularly noted for their focus on practical coding solutions, providing substantial examples and potential optimizations.

DevOptimizeAI response can be found in the Appendix F in the Section F.2.

In the *GetAPI* response, the GPT-4 model presents a well-structured code snippet for interacting with a Cosmos DB database, measuring and logging the time taken for these

interactions.

The *GetMean* response, meanwhile, provides an optimized data fetching code, demonstrating improvements such as the "WithPartitionKey" clause and the ReadAsStreamAsync method.

The *PosAPI* response addresses a potential failure during a Cosmos DB insertion operation, offering a practical solution through detailed code.

These responses, while rich in practical coding solutions, differ from previous GPT-4 model responses in their reduced focus on high-level explanations of the code and the testing process. The comparison reveals that while the model excels at providing detailed code examples, it may need to improve the balance between practical code and high-level explanations for a comprehensive understanding.

System & User Message prompt engineering

This paragraph will look at the results of the tests performed with prompt engineering both at the user message level and at the system message level.

For a comprehensive understanding of the tests performed on these enhanced API functions, refer to Appendix F in Section F.3.

In the case of *GetAPI*, the initial issue was difficulty in data retrieval from the Cosmos DB, resulting in a 404 status code. The recommended solution goes beyond a simple explanation and provides a tangible code snippet demonstrating the implementation of a caching mechanism. This strategy significantly improves performance by minimizing the need for repeated database calls.

Similarly, for the *GetMean* function, the response includes a practical code example illustrating the use of CosmosDB's SQL Query capabilities for efficient data filtering. While the function was error-free, the high average response time was a concern. The suggested enhancements effectively address this issue, leading to improved efficiency and reduced data transfer from the CosmosDB to the function.

The *PostAPI* function's enhancements are also detailed, with a focus on caching, nuanced error handling, singleton design pattern for the database connection, and load testing. The revised code snippet demonstrates how the singleton pattern can be implemented for the Cosmos DB connection and how specialized error handling for *CosmosException* can provide deeper insights into potential issues.

These responses, therefore, not only provide advanced high-level explanations for each issue but also offer practical code examples to implement the proposed solutions.

In the Figure 5.7 it is possible to see the variation of performance before and after DevOptimizeAI advice. It is possible to see the enhance of the *GetAPI* and the grater enhance for the *GetMean*, but unfortunately for the *PostAPI* the model provide an example with a bad

connection to the DB, shown in Listing 5.1, that cause an 100% error rate.

```
1 ...
2 namespace PostAPI
3 {
4     public static class PostAPI
5     {
6         private static readonly HttpClient httpClient = new HttpClient();
7
8         //Database Settings
9         private static readonly CosmosClient cosmosClient;
10        private static readonly Microsoft.Azure.Cosmos.Container
11        ↪ cosmosContainer;
12
13        //Static constructor
14        static FiDelApp()
15        {
16            ...
17        }
18
19        [FunctionName("FiDelApp")]
20        public static async Task<IActionResult> Run(
21        ↪ [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route =
22        null)] HttpRequest req,
23        ILogger log)
24        {
25            ...
26        }
27    }
28 }
```

Listing 5.1: Error DB connectio in PostAPI

As expected the API with the greatest advantage was the most complex is with more points where you can improve (GetMean).

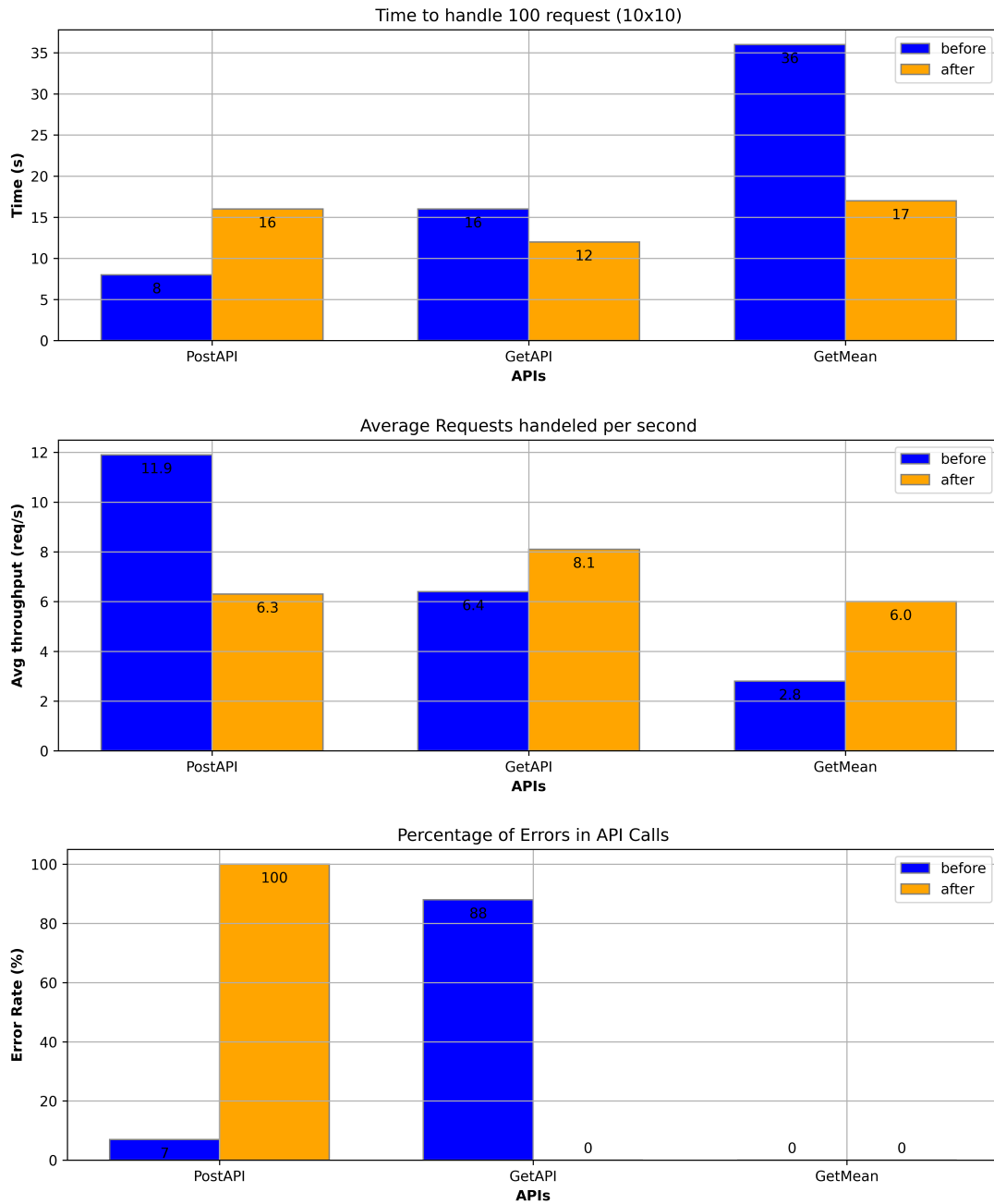


Figure 5.7: Before and after DevOptimizeAI enhancement for I/O-intensive functions

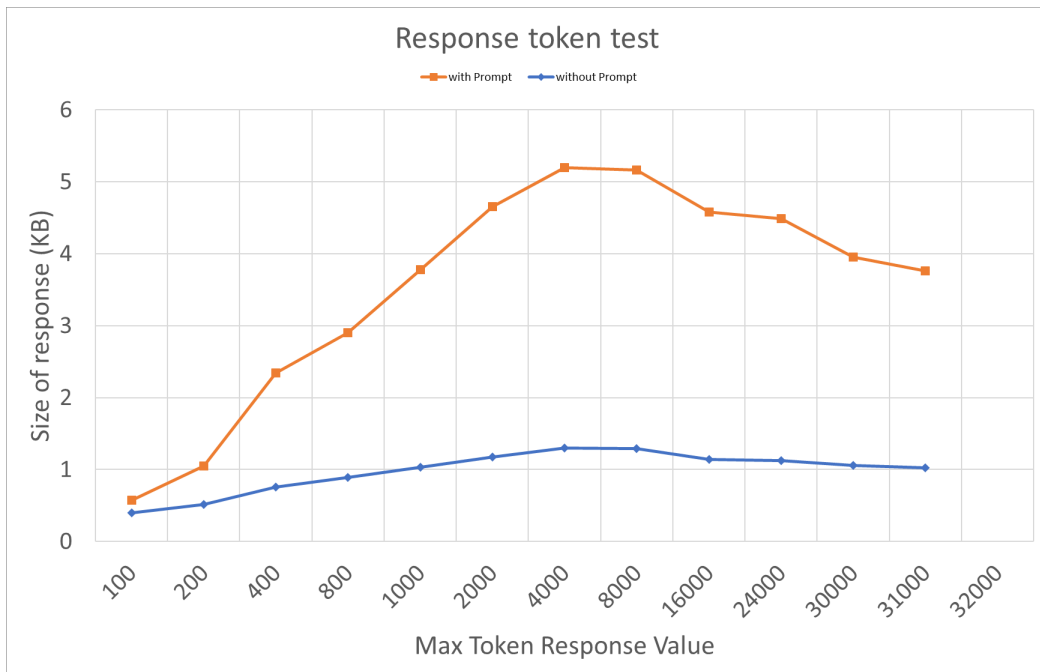


Figure 5.8: GPT response dimension varying the max token value with and without prompt engineering

Max response token test with prompt engineering

After prompt engineering, having seen the size of the response grow as the detail provided to the model increased, the Response Token test was again performed. Figure 5.8 shows the comparison between the size of the responses with prompt and without prompt. In the case of the responses with prompt, a higher expressiveness of the model is evident due to a higher level of detail or requests from DevOptimizeAI to the model.

Considerations on Prompt Engineering

The examination of DevOptimizeAI's responses to varied prompt tests has underscored the value of precision in crafting system and user prompts. The study revealed that the AI model's responses can be greatly enhanced through meticulous prompt engineering.

In particular:

- The system message level tests demonstrated the model's capability to provide valuable insights for system optimization, despite a lack of specific code examples.
- Conversely, in user message level tests, the model showcased its strength in generating detailed coding solutions, highlighting the need for balancing practical code with high-level explanations.

Test	Comments	Key Learning
System Message Prompt Engineering Test	Focused on understanding the model's performance using specification only on system message.	The model provided valuable insights and recommendations for system optimization, but didn't offer specific code examples.
User Message Prompt Engineering Test	Examined the model's responses to prompt engineering at the user message level.	The model excelled at providing detailed code examples, but needed to improve the balance between practical code and high-level explanations.
System & User Message Prompt Engineering Test	Tested prompt engineering at both the user message level and the system message level.	The unique blend of high-level explanations and practical code snippets proved particularly effective.

Table 5.6: Summary table of prompt tests

- In tests involving prompt engineering at both system and user message levels, a combination of high-level explanations and practical code examples proved most effective. This balanced approach provided a comprehensive understanding of the issues at hand and their potential solutions.

CPU-intensive test

As seen in the previous subsection, the function with the most CPU-intensive part will be the one more advantageous by the enhancement provided by DevOptimizeAI.

This section will delve into the exploration of CPU-intensive function B.4 and how DevOptimizeAI can enhance it.

The discussion of this CPU-intensive operation will be broken down into individual paragraphs for each test, enabling a comprehensive and detailed analysis.

The main tests will be:

- The original function: with some possible optimization.
- The function with a logical error in it: it will try DevOptimizeAI to understand where the function is logically wrong even if the code compile.
- The function with an infinite loop: it will try DevOptimizeAI to discover the infinite loop, fix the function and enhance it.

The Appendix G to read all the response provided by DevOptimizeAI.

An example of input file to the model for this test is provided in Appendix D.3.

Error injection test

The error injection test will involve running the function with this known error and observing if DevOptimizeAI can successfully detect and suggest a solution for it. This will demonstrate the effectiveness of DevOptimizeAI in identifying logical errors in the code, which is a critical part of code optimization.

In this particular test, we examine *CalculatePrimeNumbers* B.4 with a logical error embedded within it. The function is designed to calculate prime numbers up to a given number. However, the problem arises within the inner 'for' loop where the function tries to divide the number by zero when *i* equals to 0. This division by zero error is a common pitfall in programming and occurs when a number is divided by zero. In mathematics, division by zero is undefined, hence, it usually results in a runtime error in programming.

The line of code that introduces this error is:

- `if ((num % primes[i] == 0) || (num / (i-1) == 0))`

This test case is interesting because it provides an excellent opportunity to see if DevOptimizeAI can detect this logical error that manages to pass compilation but fails during

runtime. Being able to identify such errors can greatly help in increasing the reliability of the code and reducing debugging time.

Infinite loop test case

The infinite loop test will involve running the function with this known error and observing if DevOptimizeAI can successfully detect and propose a solution for it. This will demonstrate DevOptimizeAI's proficiency in identifying and rectifying code that could lead to infinite loops, which is a crucial aspect of code optimization.

In this test, we examine a function that inadvertently includes an infinite loop. This function is designed to calculate prime numbers up to a given number. However, the issue arises in the inner 'for' loop where it fails to increment the variable 'i'.

The line of code that introduces this error is:

- `for (int i = 0; primes[i] * primes[i] <= num;)`

In this loop, 'i' never increments, causing the loop to run indefinitely whenever the condition `primes[i] * primes[i] <= num` is met. This results in an infinite loop, one of the most common and frustrating errors in programming, which can cause the program to freeze or use excessive amounts of CPU, making it unresponsive.

This test case is intriguing because it provides an opportunity to see if DevOptimizeAI can detect this infinite loop, which otherwise might go unnoticed until runtime. Being able to identify such issues can significantly enhance the performance and reliability of the code.

CPU-intensive Test results

The results of the tests conducted on DevOptimizeAI were both revealing and promising. The tool demonstrated proficiency in identifying and rectifying common coding issues, such as logical errors and infinite loops, that can significantly affect the performance and stability of a program.

A quantitative analysis of the test results revealed that DevOptimizeAI successfully detected the errors in each scenario. In the Error Injection test, it correctly identified the division by zero error, and in the Infinite Loop test, it successfully detected the infinite loop caused by lack of incrementation.

In Figure 5.9 are presented the value that are also in Table 5.7 that represent the quantitative aspect of these test and in details:

- Original Test: This test was carried out using the original function without any injected errors. From the table, it is clear that DevOptimizeAI made significant

		Original	Error Injection	Infinite Loop
Time (sec)	Before	70.75	0.59	N/A
	After	30.30	16.7	25
Throughput (req x sec)	Before	1.414	168.5	N/A
	After	3.3	6	4.1
Errors (%)	Before	0	100	100
	After	0	0	0

Table 5.7: Comparison between tests in CPU-intensive tests part (Transposed)

improvements in the function’s performance. The time taken for execution decreased from 70.75 sec to 30.30 sec, and the throughput increased from 1.414 req/sec to 3.3 req/sec. This confirms that DevOptimizeAI was able to optimize the function effectively.

- **Error Injection Test:** In this test, a logical error was intentionally introduced into the function to test the effectiveness of DevOptimizeAI in detecting and fixing it. From the table, we can observe that before the optimization, the function had an error rate of 100%, indicating that it was failing every time. However, after the optimization, the error rate dropped to 0%, indicating that DevOptimizeAI was successful in identifying and fixing the error. Also, the time taken for execution decreased from 0.59 sec to 16.7 sec, and the throughput decreased from 168.5 req/sec to 6 req/sec. This could be due to the fact that the erroneous function was failing quickly before optimization (thus the high throughput and low time), while after the error was fixed, it took a longer time for correct execution. DevOptimizeAI not only fix the error, but also improve the function achieving the best throughput result.
- **Infinite Loop Test:** In this test, an infinite loop was intentionally introduced into the function. Before optimization, the function was not able to complete execution (hence the N/A values for time and throughput), and had an error rate of 100%. After optimization by DevOptimizeAI, the function was able to complete execution in 25 sec, with a throughput of 4.1 req/sec, and an error rate of 0%. This shows that DevOptimizeAI was successful in identifying and fixing the infinite loop.

Some consideration These tests have provided some interesting details about DevOptimizeAI capabilities in identifying and rectifying common coding errors that can severely reduce the performance and stability of a program. The tool not only successfully detected and fixed logical errors, but it also accurately identified and resolved infinite loops. In all

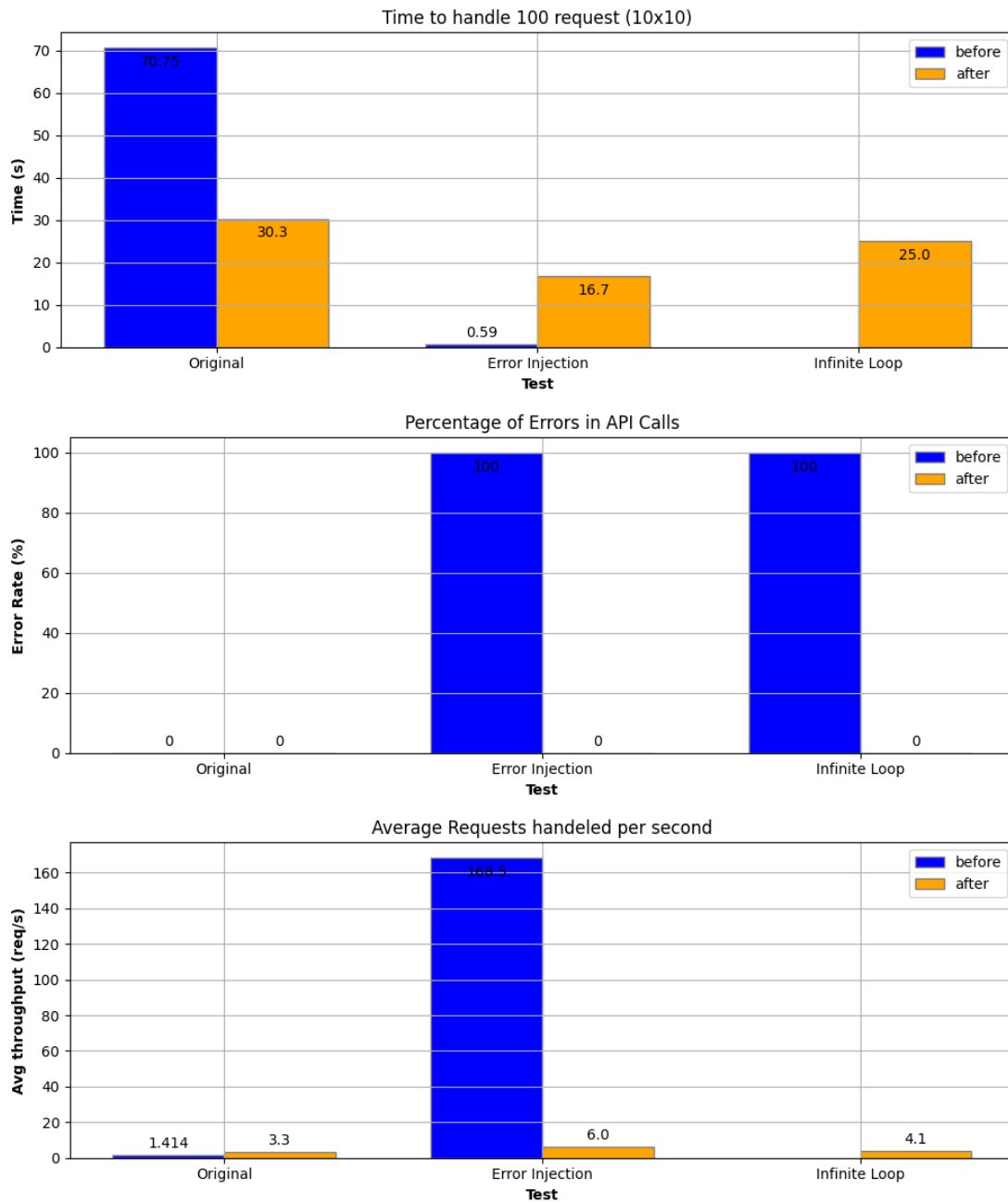


Figure 5.9: Before and after DevOptimizeAI enhancement for CPU-intensive function

the tested scenarios, DevOptimizeAI was able to improve the performance and reliability of the function, substantiating its effectiveness as a code optimization tool.

The results also indicate that DevOptimizeAI can be particularly beneficial for complex, CPU-intensive functions. By optimizing such functions, it can significantly enhance the overall efficiency and performance of the software. Furthermore, the tool's ability to detect errors that might go unnoticed until runtime can help in reducing debugging time and improving code reliability.

Behavior tests

In this subsection, the different behavior tests conducted on DevOptimizeAI are presented to go and understand how DevOptimizeAI really works and to go and unearth interesting insights. Behavior analysis will include 4 tests:

- DevOptimizeAI behavior if it know the developer's ability.
- DevOptimizeAI behavior when aided with code cues.
- DevOptimizeAI behavior if given examples of well-formulated answers.
- DevOptimizeAI behavior if reused multiple times.

An example of input file to the model for this test is provided in Appendix D.3.

DevOptimizeAI behaviour oriented to different developer types

This section will analyze the behavior of DevOptimizeAI in case a particular use is made for a specific type of developer. DevOptimizeAI response can be found in Appendix H, specifically in Section H.1. An array of tests were conducted using various system prompts and user messages, but only the two cases that most closely aligned with the established objectives have been included in the appendix. Despite the extensive testing, the results did not meet the anticipated standards.

Interestingly, when the DevOptimizeAI was tasked to provide advanced advice to a more experienced developer, it underperformed. The responses lacked tangible code examples and the advice offered was often basic. Similarly, when instructed to cater to an inexperienced or junior developer, DevOptimizeAI's performance was largely unchanged, producing advice and response messages similar to those generated without specific user details.

DevOptimizeAI seems to operate more efficiently when user specifications are kept generic, rather than furnishing it with detailed information about the user's level of experience; this could be caused by an overly specific prompt that causes difficulty for the model to find appropriate information.

DevOptimizeAI behaviour with hints in the code

This paragraph provides an evaluation of test results based on the advice model coded directly into the comments. DevOptimizeAI response can be found in Appendix H, specifically in Section H.2.

One noteworthy aspect of the test results analysis is that the suggested improvements align directly with the hints provided in the code comments, rather than offering an entirely new version. This approach emphasizes the value of embedded advice in the code and its potential for optimizing system performance.

It is possible to think of this approach as a version of Copilot that does not provide immediate code, but after a more careful analysis of even the performance of the software.

In this instance, the system managed an average of 2.6 requests per second over 100 requests, with an average server-side response time of 3589 milliseconds. While these metrics indicate that the system is functional, they also reveal areas for enhancement, specifically in reducing response times.

The primary contributors to the reduce response times were identified as the CosmosDB queries. By refining the querying strategy as suggested in the code comments, performance enhancements are achievable.

The revised function, which aligns with the provided hints, offloads data filtering tasks to CosmosDB, thus optimizing response times.

Additional enhancements can be achieved by refining the application's connection and interaction with CosmosDB, as suggested in the embedded advice. Implementing connection caching, optimizing database indexes, partitioning, and analyzing the collection RU/s and load can all contribute to performance improvements.

The potential of this approach extends beyond individual tests. It can significantly improve software development practices by providing immediate, context-specific guidance and best practices directly in the code. By following these hints, experienced developers can enhance both the efficiency and effectiveness of their code, streamline the debugging process, and ensure their systems are equipped to handle varying loads and traffic spikes.

DevOptimizeAI behaviour with previous successful messages example

In this paragraph will be presented the test done with DevOptimizeAI to understand that it is possible to improve the response of the model providing to it good example of response.

This test is set up by giving the to model *user* message and *assistant* message of different functions (e.g. the GetAPI and PostAPI request and response example). The user message, as previously stated, is the message sent from the user that query the model, the assistant message is the response example provided to the model by the developer of the model to

answer the user message. DevOptimizeAI response is included in Appendix H in Section H.3.

This experiment did not provide an extremely satisfactory result when used for a model to which prompt engineering had been made, but if the model in question does not have any type of prompt engineering can lead the model to be more accurate and efficient without having to do prompt engineering.

This approach can also be advantageous in models where prompt engineering has been done and a specific response format is wished.

Implementation of feedback provided

This paragraph will summarize the findings from the tests conducted following the model's feedback, particularly after applying the 'hint test' suggestions. DevOptimizeAI response is included in Appendix H, more specifically in Section H.4.

The first feedback, provided after the integration of the 'hint test' suggestions, offered a valuable solution. By identifying that CosmosDB operations were being executed synchronously, potentially leading to blocking IO operations, the model recommended a shift to asynchronous programming. The revised code, utilizing tasks for parallel data processing and a lock for synchronization, provided a potentially significant enhancement in efficiency, especially when processing a large number of items.

However, the subsequent feedback messages revealed some complexities. The second feedback, while offering an insightful solution, contained some logical errors. Although it correctly identified issues such as 502 Bad Gateway and 503 Service Unavailable errors, the errors encountered were related more to the suggested modifications than to the database operations.

```
1 ...
2     var tasks = new List<Task>();
3     var sum = 0.0;
4     var count = 0;
5     var queryResultSetIterator = cosmosContainer.GetItemQueryIterator<
↪ Volcano>(queryDefinition);
6
7     while (queryResultSetIterator.HasMoreResults)
8     {
9         // Start a new task to process this set of results
10        tasks.Add(Task.Run(async () =>
11        {
12            var currentResultSet = await queryResultSetIterator.
↪ ReadNextAsync();
13
14            foreach (var v in currentResultSet)
15            {
```



```

16         if (int.TryParse(v.Elevation, out var parsedElevation))
17         {
18             Interlocked.Add(ref sum, parsedElevation);
19             Interlocked.Increment(ref count);
20         }
21         else
22         {
23             ...
24         }
25     ...

```

Listing 5.2: Error in provided code

The error is in *Interlocked.Add(ref sum, parsedElevation)*; because the *Interlocked.Add* do not work with double or float, but only with int and uint. So before moving to the third feedback iteration i fix the script as changing the devinition of sum from *var sum = 0.0* to *int sum = 0* to allow the correct execution of the program.

The third feedback, despite proposing several strategies for performance enhancement, provided a solution that contained logical errors into the usage of the asynchronous pattern that make the usage of the proposed snippet impossible.

Notably, it incorrectly used an integer parameter instead of a double. This highlights the importance of thorough code review and testing, as even seemingly beneficial solutions can introduce new errors if not correctly implemented.

Considerations on Behavior Tests

Behavior tests on DevOptimizeAI revealed key insights on its function under different conditions. Four main tests were conducted: varying developer abilities, code cues, well-formulated answer examples, and multiple requests.

Following are the main insight:

- model underperformed when catering to advanced developers and didn't significantly adjust when addressing inexperienced developers. The model seemed to operate best with generic user specifications rather than detailed user experience levels.
- DevOptimizeAI's responses are aligned directly with the hints provided in the code comments, emphasizing the value of embedded advice for performance optimization (delayed Copilot).
- The test with assistant configuration didn't significantly improve the model's performance when prompt engineering was applied. However, it was noted that providing successful response examples could enhance the model's performance when prompt engineering was not applied.

Test	Comments	Key Learning
DevOptimizeAI behaviour oriented to different developer types	Analyzed the model's performance when tasked to cater to developers of varying experience levels.	The model operates more efficiently when user specifications are kept generic.
DevOptimizeAI behaviour with hints in the code	Evaluated the model's responses based on advice coded directly into comments (delayed Copilot).	Embedded advice in code comments can guide DevOptimizeAI towards optimized response.
DevOptimizeAI behaviour with previous successful messages example	Examined how providing the model with successful response examples can influence its performance.	The model's performance can be enhanced by providing it with examples of successful responses, especially when prompt engineering is not applied.
Implementation of feedback provided	Summarized the findings from the tests conducted following the model's feedback.	The value of iterative enhancement and rigorous testing in software development was underscored.

Table 5.8: Summary table of behavior tests

- The implementation of feedback from the DevOptimizeAI revealed the potential for logical errors in the proposed solutions.

ChatGPT vs DevOptimizeAI

The last test delves into a comparison with the usage of DevOptimizeAI and the usage of the only chatGPT.

The purpose of this analysis is to scrutinize the respective strengths and weaknesses of these approach, thereby gaining valuable insights into their potential applications. This analysis focuses on two AI-based systems: DevOptimizeAI and ChatGPT.

This test compare the enhanced function provided by these methods.

For DevOptimizeAI will be used the GetMean with its enhanced version provided in the test with system and user prompt, meanwhile, for the chatGPT it will be used a new generated function and its enhanced version again from ChatGPT.

The GPT response is included in Appendix I.

An example of input file for DevOptimizeAI for this test is provided in Appendix D.3.

Here's a comparison of the two enhanced implementations:

- **Dependency Injection:** The GPT implementation uses Dependency Injection to instantiate the DocumentClient. This allows for better connection management and performance. The DevOptimizeAI implementation initializes the CosmosClient directly within the implementation class.
- **Cosmos DB SDK:** The GPT implementation uses the older DocumentDB SDK (Microsoft.Azure.Documents.Client) while the DevOptimizeAI implementation uses the newer Cosmos DB SDK (Microsoft.Azure.Cosmos).
- **Error Handling:** The DevOptimizeAI implementation has more extensive error handling, including a try-catch block around the main logic and logging of errors.
- **Data Model:** The DevOptimizeAI implementation uses a specific data model (Volcano and Location classes). The GPT implementation does not assume a specific data model and operates on dynamic data.
- **Query Parameter:** The GPT implementation uses a route parameter to get the country, while the DevOptimizeAI implementation uses a query parameter.
- **Authorization Level:** The GPT implementation uses AuthorizationLevel.Function, which means the implementation requires the implementation key in the request. The DevOptimizeAI implementation uses AuthorizationLevel.Anonymous, which means the implementation can be triggered without authentication.

- **Average Calculation:** The DevOptimizeAI implementation calculates the sum and count during the Cosmos DB data retrieval, and then calculates the average. The GPT implementation retrieves all the elevations, stores them in a list, and then calculates the average. The approach in the DevOptimizeAI implementation can be more efficient, especially if the data set is large.
- **Elevation Parsing:** The DevOptimizeAI implementation expects Elevation as a string and attempts to parse it to an integer. The GPT implementation expects Elevation as an integer. If the data in Cosmos DB has Elevation as a string, the GPT implementation will not work correctly.
- **Environment Variables:** The DevOptimizeAI implementation retrieves the Cosmos DB details from environment variables, which can be more secure and flexible than hard-coding them in the implementation code as in the GPT implementation.
- **Null Check:** The DevOptimizeAI implementation checks if the elevation can be parsed to int and if not, it gives a warning and ignores the elevation. The GPT implementation does not have this check.

Both implementations do the job, but the choice between them depends on the specific requirements, the data model in Cosmos DB, and the preferred coding and error handling practices.

In terms of correctness:

- Both implementations are correct as they are able to query data from Cosmos DB and calculate the mean of the elevations.
- The DevOptimizeAI implementation uses a specific data model (Volcano and Location classes). If the data in Cosmos DB matches this model, this implementation will work correctly. If the data model is different, this implementation will fail.
- The GPT implementation does not assume a specific data model and operates on dynamic data. This makes the implementation more flexible but also requires that the data in Cosmos DB has Elevation as an integer. If Elevation is a string or other type, this implementation will fail.
- The DevOptimizeAI implementation expects Elevation as a string and attempts to parse it to an integer. If the parsing fails, it logs a warning and ignores the elevation. This makes the implementation more robust against data inconsistencies.

In terms of performance:

- The GPT implementation uses Dependency Injection to manage the DocumentClient, which can lead to better performance due to better connection management.
- The DevOptimizeAI implementation calculates the sum and count during the Cosmos DB data retrieval, and then calculates the average. This can be more efficient than the GPT implementation, which retrieves all the elevations, stores them in a list, and then calculates the average. This is especially true if the data set is large.
- The DevOptimizeAI implementation uses the newer Cosmos DB SDK (Microsoft.Azure.Cosmos), which has performance improvements and new features compared to the older DocumentDB SDK used in the GPT implementation.
- The GPT implementation uses a cross-partition query, which can be more expensive in terms of Request Unit (RU) charge compared to a single-partition query. If the Country is the partition key, a single-partition query could be used for better performance.

In conclusion, while both implementations are correct given the right conditions, the DevOptimizeAI implementation is likely to have better performance due to the reasons mentioned above. However, the actual performance can depend on various factors such as the data size, the data model, the partitioning strategy, and the frequency of implementation execution.

Chapter 6

Conclusions and future works

In this last chapter will be presented the conclusion and the possible future works about this prototype and this final project.

6.1 Conclusions

This thesis has successfully navigated the intersection of DevOps, Generative AI, and Performance Testing Tools, resulting in the creation of DevOptimizeAI. This tool brings together the efficiency of DevOps, the innovative capabilities of Generative AI, and the precision of Performance Testing Tools to improve Azure Functions software development.

This thesis has presented DevOptimizeAI, a novel tool designed to enhance the efficiency and quality of Azure Functions within a DevOps environment. The tool, which combines Generative Pretrained Transformer (GPT) models and Apache JMeter, offers an automated solution for code optimization, ensuring superior software quality and operational efficiency.

DevOptimizeAI operates within the CI/CD pipeline allowing it to analyze the software code and test results in real-time. This seamless integration is guided by a comprehensive design and analysis process that incorporates both functional and non-functional aspects.

The high-level design of DevOptimizeAI involves the creation of a configuration form and two YAML templates for the JMeter and the OpenAI stages, with each component having a specific role in the system. The configuration form is a crucial part of the user interface, providing developers with an easy-to-use platform to input their requirements and control the testing process. The JMeter stage performs performance testing on the deployed application and test formatting for better suit the GPT model integration, while the OpenAI stage uses a GPT model to analyze the test results and the Azure Function

code in conjunction, generating a detailed analysis report highlighting bottleneck, logical error and areas where improvement are possible with, also, implementation guidance and explanation.

The primary aim of this thesis was to evaluate the potential benefits of using GPT models and Apache JMeter in a DevOps scenario of REST API development via Azure Functions. The results show that DevOptimizeAI can indeed improve the software development process, leading to more efficient, reliable, and robust web applications and APIs.

DevOptimizeAI was put to test using three specific Azure Function APIs (GetAPI, PostAPI, and GetMean) which were created to simulate various real-world scenarios and another function (CalculatePrimeNumbers) to simulate CPU-intensive API. The performance of these functions under different conditions was observed and analyzed, providing valuable insights into the system's capabilities and potential areas for improvement.

A first set of acceptance testing test was a crucial part of the evaluation process, focusing on validating the interaction between Azure Functions, the GPT model, and JMeter.

Then more specific validation testing were conducted on prompt engineering and on a CPU-Intensive function. Others behavioural test were conducted in different condition: assessing the system's performance as it catered to developers of varying experience levels, handled different types of code cues, and responded to multiple requests.

The results of these tests underscored the importance of prompt engineering in optimizing the model's responses. It was observed that the model operates more efficiently when user specifications are kept generic but both system and user specification message are provided. The practice of embedding advice in code comments was found to guide developers towards optimized system performance, and providing the model with examples of successful responses could enhance its performance.

A comparison between DevOptimizeAI and ChatGPT revealed that while both tools can provide context-specific advice and code improvement suggestions, they have different strengths and weaknesses. The choice between them depends on the specific requirements of the project, the data model, and the preferred coding and error handling practices.

Furthermore, this research contributes to the ongoing discourse on software development best practices, indicating that innovative solutions like DevOptimizeAI can mitigate the challenges faced by large development teams, particularly in maintaining high-quality software.

In conclusion, the extensive testing and experiments conducted on DevOptimizeAI have provided valuable insights into its capabilities and limitations. These findings pave the way for future enhancements and improvements, making DevOptimizeAI a promising tool for software development teams working in a DevOps setting for enhancing the software development process in an Azure Function and Azure DevOps development environment.

6.2 Future works

Future works for DevOptimizeAI and the project can focus on extending the functionality and exploring new use cases for AI models in the DevOps scenario. The following are some possible extensions and future works for the prototype and the project:

- Integration with other AI models: DevOptimizeAI currently utilizes the OpenAI API for its AI capabilities. Future works can explore the integration of other AI models to enhance the prototype's capabilities and provide developers with a wider range of recommendations.
- Integration with other DevOps tools: DevOptimizeAI currently integrates with Azure DevOps pipelines. Future works can explore the integration of the prototype with other DevOps tools to provide developers with a more comprehensive set of recommendations for system optimization.
- Improved handling of errors: The prototype currently identifies bottlenecks in the software and provides feedback to developers and also provide correction to logical error. Future works can focus on developing models that can diagnose and recommend fixes for syntax errors in the code.
- Integration with other types of applications: DevOptimizeAI's functionality and effectiveness are currently bound to web APIs written in Azure Function style. Future works can explore the integration of the prototype with other types of applications to increase its applicability.
- Exploration of new use cases: Future works can explore new use cases for AI models in the DevOps scenario. For example, AI models can be used to optimize database configurations or provide recommendations for optimizing cloud infrastructure.

In light of recent advancements, such as the introduction of customizable versions of ChatGPT (called GPTs) by OpenAI, the future of DevOptimizeAI could involve the integration of these tailored AI models. This would enable more personalized and specific AI applications in the DevOps scenario.

Additionally, OpenAI's incoming GPT Store, which will feature creations from verified builders, presents a potential platform for sharing and monetizing optimized AI models for DevOps. This could lead to a broader range of AI tools and solutions available for integration with DevOptimizeAI, improving its capabilities and potential use cases.

List of Tables

2.1	Most popular commercial LLM comparison	19
5.1	Comprehensive Overview of Tests Conducted	65
5.2	Comparison of GPT-3 and GPT-4 Tests with 4k max token response . . .	76
5.3	Response token test table	78
5.4	Summary of load test results	80
5.5	Summary table of integration tests	81
5.6	Summary table of prompt tests	89
5.7	Comparison between tests in CPU-intensive tests part (Transposed)	92
5.8	Summary table of behavior tests	98

List of Figures

2.1	DevOps	10
2.2	ChatGPT Methodology Diagram	18
2.3	Rate of incorrect behavior on sensitive and disallowed prompts. Lower values are better. GPT-4 RLHF exhibits significantly lower incorrect behavior rates compared to prior models [32].	30
3.1	Data flow diagram	37
3.2	Entity-Relationship requirements diagram	38
3.3	High level sequence diagram	40
3.4	Architecture - system overview	43
3.5	High-level pipeline design	44
3.6	High-level DevOptimizeAI design	45
4.1	Configuration Form	52
4.2	JMeter and OpenAI stages flow chart	56
5.1	GetAPI - API Flow Chart	68
5.2	PostAPI - API Flow Chart	70
5.3	GetMean - API Flow Chart	72
5.4	CalculatePrimeNumbers - API Flow Chart	74
5.5	GPT response dimension varying the max token value	77
5.6	Load Test Graph	79

5.7	Before and after DevOptimizeAI enhancement for I/O-intensive functions .	87
5.8	GPT response dimension varying the max token value with and without prompt engineering	88
5.9	Before and after DevOptimizeAI enhancement for CPU-intensive function .	93

Bibliography

- [1] GitHub Copilot. Website. URL: <https://github.com/features/copilot>.
- [2] Openai text generation guide, 2022. [Online; accessed on Date of access]. URL: <https://platform.openai.com/docs/guides/text-generation>.
- [3] Rabiya Abbas, Zainab Sultan, and Shahid Nazir Bhatti. Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege. In *2017 International Conference on Communication Technologies (ComTech)*, Bahria University Islamabad, Pakistan, 2017. IEEE.
- [4] Mamdouh Alenezi, Mohammad Zarour, and Mohammad Akour. Can artificial intelligence transform devops?, 2022. URL: <https://arxiv.org/abs/2206.00225>.
- [5] S. Alla and S. K. Adari. What is mlops? In *Beginning MLOps with MLFlow*, pages 79–124. Springer, 2021.
- [6] Apache Software Foundation. Apache jmeter - Apache JMeter™, 2023. URL: <https://jmeter.apache.org/>.
- [7] M. Baldwin. Azure key vault documentation, 2023. [Online; accessed 03-January-2023]. URL: <https://learn.microsoft.com/en-us/azure/key-vault/>.
- [8] S. Chamoli. Implementing ai for improved performance testing, April 24 2023. URL: <https://www.openxcell.com/blog/implementing-ai-for-improved-performance-testing/>.
- [9] S. Chamoli. Manual testing vs automation testing: Which one should you prefer. Online Blog Post, June 20 2023. URL: <https://www.openxcell.com/blog/manual-testing-vs-automation-testing-which-one-should-you-prefer/>.
- [10] S. Chamoli. What is api testing: Complete guide. Online Blog Post, May 18 2023. URL: <https://www.openxcell.com/blog/what-is-api-testing-complete-guide/>.

- [11] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.
- [12] Martin Ekuan. Azure subscription documentation, 2023. [Online; accessed 09-October-2023]. URL: <https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-area/resource-org-subscriptions>.
- [13] Eric Freeman. Devops, 2019. URL: <https://aws.amazon.com/devops/what-is-devops/>.
- [14] Glenn Gailey. Azure functions documentation, 2023. [Online; accessed 24-May-2023]. URL: <https://learn.microsoft.com/en-us/azure/azure-functions/>.
- [15] N. Gift and A. Deza. *Practical MLOps*. O’Reilly Media, Inc., 2021.
- [16] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Waterville, Maine, USA, 2022. Colby College. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9793778>.
- [17] Mike Jacobs. Ci/cd baseline architecture with azure pipelines. Microsoft Learn, August 2023.
- [18] Mike Jacobs, Theano Petersen, Ed Kaim, Steve Danielson, and Dan Hellem. What is devops?, 01 2023. URL: <https://learn.microsoft.com/en-us/devops/what-is-devops>.
- [19] Eirini Kalliamvakou. Research: quantifying github copilot’s impact on developer productivity and happiness. GitHub Blog, September 7 2022. URL: <https://github.blog/2022-09-07-research-quantifying-github%2Dcopilots-impact-on-developer-productivity-and-happiness/>.
- [20] Nigel Kersten, Alanna Brown, Jez Humble, Dr. Nicole Forsgren, and Gene Kim. 2016 State of DevOps Report. Technical report, Puppet + DORA, 2016. URL: <http://devops-research.com>.
- [21] Eran Kinsbruner. How artificial intelligence (ai) and machine learning are changing devops, 2022. Accessed: May 01, 2022. URL: <https://enterpriseproject.com/>.
- [22] Jennifer Kulla-Mader. What is azure pipelines? - azure pipelines, December 2023. URL: <https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>.

- [23] Julia Kulla-Mader. Azure pipelines documentation, 2023. [Online; accessed 12-April-2023]. URL: <https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines>.
- [24] Deepali Medchal. Chatgpt vs google bard vs bing chat, 2023. [Online; accessed on Date of access]. URL: <https://www.openxcell.com/blog/chatgpt-vs-google-bard-vs-bing-chat/>.
- [25] Mijacobs. What is devops? - azure devops, 2023. URL: <https://learn.microsoft.com/en-us/devops/what-is-devops>.
- [26] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023. URL: <https://www.sciencedirect.com/science/article/pii/S0164121223001292>, doi:10.1016/j.jss.2023.111734.
- [27] Roopesh Nair. Service connection in azure pipeline, 2023. [Online; accessed 28-November-2023]. URL: <https://learn.microsoft.com/en-us/azure/devops/pipelines/library/service-endpoints?view=azure-devops&tabs=yaml>.
- [28] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot’s code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, Edmonton, AB, Canada, 2022. University of Alberta. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9796235>.
- [29] OpenAI. About OpenAI. <https://openai.com/about>, 2015–2023. OpenAI © 2015–2023.
- [30] OpenAI. ChatGPT Blog Post. <https://openai.com/blog/chatgpt>, November 2022. Accessed on July 6, 2023.
- [31] OpenAI. GPT-4. <https://openai.com/research/gpt-4>, March 14 2023.
- [32] OpenAI. GPT-4 Technical Report. <https://doi.org/10.48550/arXiv.2303.08774>, March 15 2023. Submitted on 15 Mar 2023 (v1), last revised 27 Mar 2023 (this version, v3). arXiv:arXiv:2303.08774.
- [33] Mohammad Rizky Pratama and Dana Sulistiyo Kusumo. Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing. In *2021 9th International Conference on Information and Communication Technology (ICoICT)*, pages 230–235, 2021. doi:10.1109/ICoICT52021.2021.9527496.
- [34] I. Pölöskei. Mlops approach in the cloud-native data pipeline design. *Acta Technica Jaurinensis*, 15(1):1–6, 2022.

- [35] M. Rajkumar, A.K. Pole, V.S. Adige, and P. Mahanta. Devops culture and its impact on cloud delivery and software development. In *2016 International Conference on Advances in Computing Communication & Automation (ICACCA)(Spring)*, pages 1–6, 2016.
- [36] McLean Schofield. Microsoft azure documentation, 2022. [Online; accessed 18-October-2022]. URL: <https://learn.microsoft.com/en-us/azure/developer/intro/azure-developer-overview>.
- [37] Kalaivani Sundararajan. What is generative ai? a brief overview, 2023. [Online; accessed on Date of access]. URL: <https://www.openxcell.com/blog/what-is-generative-ai-brief-overview/>.
- [38] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas. Mlops–definitions, tools and challenges, 2022. URL: <https://arxiv.org/abs/2201.00162>.
- [39] D. A. Tamburri. Sustainable mlops: Trends and challenges. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 17–23. IEEE, 2020.
- [40] M. Treveil, N. Omont, C. Stenac, K. Lefevre, D. Phan, J. Zentici, A. Lavoillotte, M. Miyazaki, and L. Heidmann. *Introducing MLOps*. O’Reilly Media, 2020.
- [41] S. Verdi. Inside github: Working with the llms behind github copilot. The GitHub Blog, September 29 2023. URL: <https://github.blog/2023-05-17-inside-github-working-with-the-llms-behind-github-copilot/>.
- [42] Chase Wilson. Azure devops documentation, 2023. [Online; accessed 10-October-2023]. URL: <https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?>

Appendix A

Template YAML code

A.1 JMeter YAML code

```
1 parameters:
2   - name: threads
3     type: string
4     default: '10'
5   - name: rampup
6     type: string
7     default: '1'
8   - name: loops
9     type: string
10    default: '10'
11
12 jobs:
13 - job: Test
14   displayName: Test
15   pool:
16     vmImage: 'windows-2019'
17
18   steps:
19   - task: PowerShell@2
20     displayName: 'Install Apache JMeter'
21     inputs:
22       targetType: 'inline'
23       script: |
```



```

24     $jmeterZipUrl = 'https://archive.apache.org/dist/
      ↪ jmeter/binaries/apache-jmeter-5.5.zip'
25     $jmeterZipPath = '$(System.DefaultWorkingDirectory)/
      ↪ jmeter.zip'
26     $jmeterExtractPath = '$(System.DefaultWorkingDirectory
      ↪ )/jmeter'
27
28     Invoke-WebRequest -Uri $jmeterZipUrl -OutFile
      ↪ $jmeterZipPath
29     Expand-Archive -Path $jmeterZipPath -DestinationPath
      ↪ $jmeterExtractPath -Forc
30
31 - task: PowerShell@2
32     displayName: 'Set parameters runtime'
33     inputs:
34         targetType: 'filePath'
35         filePath: './scripts/set_parameters.ps1'
36         arguments: '-threads ${{ parameters.threads }} -rampup $
      ↪ {{ parameters.rampup }} -loops ${{ parameters.
      ↪ loops }}'
37
38 - script: |
39     $(System.DefaultWorkingDirectory)/jmeter/apache-jmeter
      ↪ -5.5/bin/jmeter -n -t $(System.
      ↪ DefaultWorkingDirectory)/*.jmx -l $(System.
      ↪ DefaultWorkingDirectory)/test_summary.jtl > $(
      ↪ System.DefaultWorkingDirectory)/summary.txt
40     displayName: 'Run JMeter Test'
41
42 - task: PythonScript@0
43     displayName: 'Filter and concatenate test results'
44     inputs:
45         scriptSource: 'filePath'
46         scriptPath: './scripts/filter_concatenate_test_results.
      ↪ py'
47         arguments: '--summary $(System.DefaultWorkingDirectory)/
      ↪ summary.txt --test_summary $(System.
      ↪ DefaultWorkingDirectory)/test_summary.jtl'
48
49 - task: PublishPipelineArtifact@1
50     inputs:

```

```
51 |     targetPath: '$(System.DefaultWorkingDirectory)/  
    |         ↳ TestResults.txt'  
52 |     artifact: 'JTLResults'
```

Listing A.1: JMeter YAML code

A.2 OpenAi YAML code

```
1 | parameters:  
2 |   - name: tokens  
3 |     type: string  
4 |     default: '4000'  
5 |   - name: temperature  
6 |     type: string  
7 |     default: '0.7'  
8 |   - name: model  
9 |     type: string  
10 |    default: 'GPT4'  
11 |   - name: workingDir  
12 |     type: string  
13 |     default: '$(System.DefaultWorkingDirectory)'  
14 |   - name: caption  
15 |     type: string  
16 |     default: 'Checkout our suggestions!'  
17 |   - name: chatid  
18 |     type: string  
19 |     default: '549538016'  
20 |   - name: azSubscription  
21 |     type: string  
22 |     default: ''  
23 |  
24 | jobs:  
25 | - job: OpenAi  
26 |   displayName: OpenAi  
27 |   pool:  
28 |     vmImage: 'windows-2019'  
29 |  
30 |   steps:  
31 | - task: DownloadPipelineArtifact@2
```

```

32     displayName: 'Download JTL Results Artifact'
33     inputs:
34         artifactName: 'JTLResults'
35         targetPath: '$(System.DefaultWorkingDirectory)'
36
37 - task: AzureKeyVault@2
38     inputs:
39         azureSubscription: '${{ parameters.azureSubscription }}'
40         KeyVaultName: 'DevEmpower'
41         SecretsFilter: '${{ parameters.model }}-KEY,${{
42             ↪ parameters.model }}-URL'
43         RunAsPreJob: false
44
45 - task: PowerShell@2
46     displayName: 'Call ChatGPT API'
47     inputs:
48         targetType: 'filePath'
49         filePath: './scripts/call_chat_gpt_api.ps1'
50         arguments: '-jtlResultsPath $(System.
51             ↪ DefaultWorkingDirectory)/TestResults.txt -
52             ↪ functionCodePath ${{ parameters.workingDir }}/*.cs
53             ↪ -apiKey ${{ parameters.model }}-KEY -apiUrl ${{
54             ↪ parameters.model }}-URL -temperature ${{
55             ↪ parameters.temperature }} -tokens ${{ parameters.
56             ↪ tokens }}'
57
58 - task: UsePythonVersion@0
59     inputs:
60         versionSpec: '3.x'
61         addToPath: true
62         architecture: 'x64'
63
64 - script: |
65     python -m pip install --upgrade pip
66     pip install requests
67     displayName: 'Install requests'
68
69 - task: PythonScript@0
70     inputs:
71         scriptSource: 'filePath'
72         scriptPath: './scripts/send_to_telegram.py'

```

66

```
arguments: '-bot_token 6096186625:  
  ↪ AAHcTptWfiK5CIamSEcmjEtpfJA1oiKGR_Q -chat_id ${{  
  ↪ parameters.chatid }} -document_paths [$(System.  
  ↪ DefaultWorkingDirectory)/generatedText.md, $(  
  ↪ System.DefaultWorkingDirectory)/userContent.txt] -  
  ↪ caption ${{ parameters.caption }}'
```

Listing A.2: OpenAi YAML code

Appendix B

C# function code

B.1 GetMean code

```
1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Collections.Generic;
15 using System.Collections.Concurrent;
16 using System.Linq;
17
18 namespace GetMean
19 {
20     public static class GetMean
21     {
22         private static readonly string cosmosDbEndpoint = Environment.
↪ GetEnvironmentVariable("DBENDPOINT", EnvironmentVariableTarget.
↪ Process);
23         private static readonly string cosmosDbKey = Environment.
↪ GetEnvironmentVariable("DBKEY", EnvironmentVariableTarget.Process);
24         private static readonly string databaseId = Environment.
↪ GetEnvironmentVariable("DBID", EnvironmentVariableTarget.Process);
```

```

25     private static readonly string containerId = Environment.
↳ GetEnvironmentVariable("CONTAINERID", EnvironmentVariableTarget.
↳ Process);
26     private static readonly CosmosClient cosmosClient = new
↳ CosmosClient(cosmosDbEndpoint, cosmosDbKey);
27     private static readonly Microsoft.Azure.Cosmos.Container
↳ cosmosContainer = cosmosClient.GetContainer(databaseId, containerId
↳ );
28
29     [FunctionName("GetMean")]
30     public static async Task<IAActionResult> Run(
31     [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
↳ null)] HttpRequest req,
32     ILogger log)
33     {
34         ///  

35         try
36         {
37             string country = req.Query["Country"];
38
39             // Validate that the country is provided
40             if (string.IsNullOrEmpty(country))
41             {
42                 return new BadRequestObjectResult("The 'Country'
↳ query parameter is required.");
43             }
44
45             FeedIterator<Volcano> queryResultSetIterator =
↳ cosmosContainer.GetItemQueryIterator<Volcano>();
46             double sum = 0;
47             int count = 0;
48
49             while (queryResultSetIterator.HasMoreResults)
50             {
51                 FeedResponse<Volcano> currentResultSet = await
↳ queryResultSetIterator.ReadNextAsync();
52
53                 foreach (Volcano volcano in currentResultSet)
54                 {
55                     if (volcano.Country == country)
56                     {
57                         if (int.TryParse(volcano.Elevation, out int
↳ parsedElevation))
58                         {
59                             sum += parsedElevation;
60                             count++;
61                         }
62                         else
63                         {

```

```

64         log.LogWarning($"Could not parse
↪ elevation '{volcano.Elevation}' for volcano '{volcano.VolcanoName
↪ }'. This elevation will be ignored.");
65     }
66 }
67 }
68 }
69
70
71 // If no volcanoes found, return not found
72 if (count == 0)
73 {
74     return new NotFoundResult();
75 }
76
77 // Calculate the mean elevation
78 double meanElevation = Math.Round(sum / count, 3);
79
80
81     return new OkObjectResult(meanElevation);
82 }
83 catch (Exception ex)
84 {
85     log.LogError(ex, "An error occurred while retrieving data
↪ from Cosmos DB.");
86     return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
87 }
88 }
89 /**/
90 }
91 public class Volcano
92 {
93     public string VolcanoName { get; set; }
94     public string Country { get; set; }
95     public string Region { get; set; }
96     public Location Location { get; set; }
97     public string Elevation { get; set; }
98     public string Type { get; set; }
99     public string Status { get; set; }
100    public string LastKnownEruption { get; set; }
101    public string Id { get; set; }
102 }
103
104 public class Location
105 {
106     public string Type { get; set; }
107     public List<double> Coordinates { get; set; }
108 }

```

Listing B.1: GETMean

B.2 GetAPI code

```
1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Collections.Generic;
15 using System.Collections.Concurrent;
16
17 namespace GetAPI
18 {
19     public static class GetAPI
20     {
21         private static readonly string cosmosDbEndpoint = Environment.
22         ↪ GetEnvironmentVariable("DBENDPOINT", EnvironmentVariableTarget.
23         ↪ Process);
24         private static readonly string cosmosDbKey = Environment.
25         ↪ GetEnvironmentVariable("DBKEY", EnvironmentVariableTarget.Process);
26         private static readonly string databaseId = Environment.
27         ↪ GetEnvironmentVariable("DBID", EnvironmentVariableTarget.Process);
28         private static readonly string containerId = Environment.
29         ↪ GetEnvironmentVariable("CONTAINERID", EnvironmentVariableTarget.
30         ↪ Process);
31         private static readonly CosmosClient cosmosClient = new
32         ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey);
33         private static readonly Microsoft.Azure.Cosmos.Container
34         ↪ cosmosContainer = cosmosClient.GetContainer(databaseId, containerId
35         ↪ );
36
37         [FunctionName("GetWebAPI")]
38         public static async Task<IActionResult> Run(
39             [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
40             ↪ null)] HttpRequest req,
```



```

31         ILogger log)
32     {
33         try
34         {
35             string id = req.Query["id"];
36
37             // Validate that the ID is provided
38             if (string.IsNullOrEmpty(id))
39             {
40                 return new BadRequestObjectResult("The 'id' query
↪ parameter is required.");
41             }
42
43             // Retrieve the item from Cosmos DB
44             var response = await cosmosContainer.ReadItemAsync<
↪ RequestDocument>(id, new PartitionKey(id));
45
46             return new OkObjectResult(response.Resource);
47         }
48         catch (CosmosException ex) when (ex.StatusCode == System.Net.
↪ HttpStatusCode.NotFound)
49         {
50             return new NotFoundResult();
51         }
52         catch (Exception ex)
53         {
54             log.LogError(ex, "An error occurred while retrieving data
↪ from Cosmos DB.");
55             return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
56         }
57     }
58 }
59
60 public class RequestDocument
61 {
62     public string id { get; set; }
63     public string categoryId { get; set; }
64     public string categoryName { get; set; }
65     public string sku { get; set; }
66     public string name { get; set; }
67     public string description { get; set; }
68     public double Price { get; set; }
69     public List<Tag> tags { get; set; }
70 }
71
72 public class Tag
73 {
74     public string id { get; set; }

```

```

75     public string name { get; set; }
76 }
77 }

```

Listing B.2: GetAPI

B.3 PostAPI code

```

1  using System;
2  using System.IO;
3  using System.Threading.Tasks;
4  using Microsoft.AspNetCore.Mvc;
5  using Microsoft.Azure.WebJobs;
6  using Microsoft.Azure.WebJobs.Extensions.Http;
7  using Microsoft.AspNetCore.Http;
8  using Microsoft.Extensions.Logging;
9  using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Collections.Generic;
15
16 namespace PostAPI
17 {
18
19
20     public static class PostAPI
21     {
22         private static readonly HttpClient httpClient = new HttpClient();
23         private static readonly string cosmosDbEndpoint = Environment.
↪ GetEnvironmentVariable("DBENDPOINT", EnvironmentVariableTarget.
↪ Process);
24         private static readonly string cosmosDbKey = Environment.
↪ GetEnvironmentVariable("DBKEY", EnvironmentVariableTarget.Process);
25         private static readonly string databaseId = Environment.
↪ GetEnvironmentVariable("DBID", EnvironmentVariableTarget.Process);
26         private static readonly string containerId = Environment.
↪ GetEnvironmentVariable("CONTAINERID", EnvironmentVariableTarget.
↪ Process);
27         private static readonly CosmosClient cosmosClient = new
↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey);
28         private static readonly Microsoft.Azure.Cosmos.Container
↪ cosmosContainer = cosmosClient.GetContainer(databaseId, containerId
↪ );
29

```

```

30     [FunctionName("FiDelApp")]
31     public static async Task<IActionResult> Run(
32         [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route =
↪ null)] HttpRequest req,
33         ILogger log)
34     {
35         try
36         {
37             string requestBody = await new StreamReader(req.Body).
↪ ReadToEndAsync();
38
39             // Deserialize the JSON request body
40             var requestDocument = JsonConvert.DeserializeObject<
↪ RequestDocument>(requestBody);
41
42             // Validate that the ID is provided
43             if (string.IsNullOrEmpty(requestDocument.id))
44             {
45                 return new BadRequestObjectResult("The 'id' property
↪ is required.");
46             }
47
48             // Insert the JSON document into Cosmos DB
49             var response = await cosmosContainer.CreateItemAsync(
↪ requestDocument);
50
51             return new OkObjectResult("Data inserted successfully!
↪ Well Done! (12/07/2023)");
52         }
53         catch (Exception ex)
54         {
55             log.LogError(ex, "An error occurred while inserting data
↪ into Cosmos DB.");
56             return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
57         }
58     }
59 }
60 public class RequestDocument
61 {
62     public string id { get; set; }
63     public string categoryId { get; set; }
64     public string categoryName { get; set; }
65     public string sku { get; set; }
66     public string name { get; set; }
67     public string description { get; set; }
68     public double Price { get; set; }
69     public List<Tag> tags { get; set; }
70 }

```

```

71 public class Tag
72 {
73     public string id { get; set; }
74     public string name { get; set; }
75 }
76 }

```

Listing B.3: PostAPI

B.4 CalculatePrimeNumbers code

```

1 using System;
2 using System.Net;
3 using Microsoft.AspNetCore.Mvc;
4 using Microsoft.Extensions.Primitives;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using System.Collections.Generic;
9 using System.IO;
10 using Newtonsoft.Json;
11
12 public static class CalculatePrimeNumbers
13 {
14     [FunctionName("CalculatePrimeNumbers")]
15     public static async Task<IActionResult> Run(
16         [HttpTrigger(AuthorizationLevel.Function, "get", Route = null)]
17         ↪ HttpRequest req, ILogger log)
18     {
19         log.LogInformation("C# HTTP trigger function processed a request.
20         ↪ ");
21
22         string number = req.Query["number"];
23         string requestBody = await new StreamReader(req.Body).
24         ↪ ReadToEndAsync();
25         dynamic data = JsonConvert.DeserializeObject(requestBody);
26         number = number ?? data?.number;
27
28         if (int.TryParse(number, out int num))
29         {
30             List<int> primes = GetPrimes(num);
31             return new OkObjectResult($"Prime numbers up to {num}: {
32             ↪ string.Join(", ", primes)}");
33         }
34         else
35         {

```

```

32         return new BadRequestObjectResult("Please provide a valid
↪ integer.");
33     }
34 }
35
36 private static List<int> GetPrimes(int maxNum)
37 {
38     var primes = new List<int>() { 2 };
39     for (int num = 3; num <= maxNum; num += 2)
40     {
41         bool isPrime = true;
42         for (int i = 0; primes[i] * primes[i] <= num; i++)
43         {
44             if (num % primes[i] == 0)
45             {
46                 isPrime = false;
47                 break;
48             }
49         }
50         if (isPrime)
51         {
52             primes.Add(num);
53         }
54     }
55     return primes;
56 }
57 }

```

Listing B.4: CalculatePrimeNumbers

Appendix C

JMeter Test Plans

C.1 GetMean Test plan

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.5">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan"
      ↪ testname="Test_Plan" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">>false</
        ↪ boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">>true</
        ↪ boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">>false</
        ↪ boolProp>
      <elementProp name="TestPlan.user_defined_variables"
        ↪ elementType="Arguments" guiclass="ArgumentsPanel"
        ↪ testclass="Arguments" testname="User_Defined_
        ↪ Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></
        ↪ stringProp>
    </TestPlan>
  <hashTree>
    <Arguments guiclass="ArgumentsPanel" testclass="
      ↪ Arguments" testname="User_Defined_Variables">
```

```

↪ enabled="true">
<collectionProp name="Arguments.arguments">
  <elementProp name="1" elementType="Argument">
    <stringProp name="Argument.name">1</stringProp>
    <stringProp name="Argument.value">Japan</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</stringProp
      ↪ >
  </elementProp>
  <elementProp name="2" elementType="Argument">
    <stringProp name="Argument.name">2</stringProp>
    <stringProp name="Argument.value">Italy</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</stringProp
      ↪ >
  </elementProp>
  <elementProp name="3" elementType="Argument">
    <stringProp name="Argument.name">3</stringProp>
    <stringProp name="Argument.value">India</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</stringProp
      ↪ >
  </elementProp>
</collectionProp>
</Arguments>
<hashTree/>
<RandomVariableConfig guiclass="TestBeanGUI" testclass="
  ↪ RandomVariableConfig" testname="RandomVariable"
  ↪ enabled="true">
  <stringProp name="variableName">numCountry</stringProp
    ↪ >
  <stringProp name="outputFormat"></stringProp>
  <stringProp name="minimumValue">1</stringProp>
  <stringProp name="maximumValue">3</stringProp>
  <stringProp name="randomSeed"></stringProp>
  <boolProp name="perThread">>false</boolProp>
</RandomVariableConfig>
<hashTree/>
<ThreadGroup guiclass="ThreadGroupGui" testclass="
  ↪ ThreadGroup" testname="ThreadGroup" enabled="true
  ↪ ">

```

```

<stringProp name="ThreadGroup.on_sample_error">
  ↪ continue</stringProp>
<elementProp name="ThreadGroup.main_controller"
  ↪ elementType="LoopController" guiclass="
  ↪ LoopControlPanel" testclass="LoopController"
  ↪ testname="LoopController" enabled="true">
<boolProp name="LoopController.continue_forever">
  ↪ false</boolProp>
<stringProp name="LoopController.loops">10</
  ↪ stringProp>
</elementProp>
<stringProp name="ThreadGroup.num_threads">10</
  ↪ stringProp>
<stringProp name="ThreadGroup.ramp_time">1</stringProp
  ↪ >
<boolProp name="ThreadGroup.scheduler">>false</boolProp
  ↪ >
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<boolProp name="ThreadGroup.
  ↪ same_user_on_next_iteration">>true</boolProp>
</ThreadGroup>
<hashTree>
<HTTPSamplerProxy guiclass="HttpTestSampleGui"
  ↪ testclass="HTTPSamplerProxy" testname="HTTP_
  ↪ Request" enabled="true">
<elementProp name="HTTPsampler.Arguments"
  ↪ elementType="Arguments" guiclass="
  ↪ HTTPArgumentsPanel" testclass="Arguments"
  ↪ enabled="true">
<collectionProp name="Arguments.arguments">
  <elementProp name="Country" elementType="
    ↪ HTTPArgument">
    <boolProp name="HTTPArgument.always_encode">
      ↪ false</boolProp>
    <stringProp name="Argument.value">${__V(${
      ↪ numCountry})}</stringProp>
    <stringProp name="Argument.metadata">=</
      ↪ stringProp>
    <boolProp name="HTTPArgument.use_equals">>true
      ↪ </boolProp>

```



```

        <stringProp name="Argument.name">Country</
            ↳ stringProp>
    </elementProp>
</collectionProp>
</elementProp>
<stringProp name="HTTPSampler.domain">getmean.
    ↳ azurewebsites.net</stringProp>
<stringProp name="HTTPSampler.port">443</stringProp>
<stringProp name="HTTPSampler.protocol">https</
    ↳ stringProp>
<stringProp name="HTTPSampler.contentEncoding"></
    ↳ stringProp>
<stringProp name="HTTPSampler.path">/api/GetMean</
    ↳ stringProp>
<stringProp name="HTTPSampler.method">GET</
    ↳ stringProp>
<boolProp name="HTTPSampler.follow_redirects">true</
    ↳ boolProp>
<boolProp name="HTTPSampler.auto_redirects">false</
    ↳ boolProp>
<boolProp name="HTTPSampler.use_keepalive">true</
    ↳ boolProp>
<boolProp name="HTTPSampler.DO_MULTIPART_POST">>false
    ↳ </boolProp>
<stringProp name="HTTPSampler.embedded_url_re"></
    ↳ stringProp>
<stringProp name="HTTPSampler.connect_timeout"></
    ↳ stringProp>
<stringProp name="HTTPSampler.response_timeout"></
    ↳ stringProp>
</HTTPSamplerProxy>
<hashTree>
    <ResultCollector guiclass="ViewResultsFullVisualizer
        ↳ " testclass="ResultCollector" testname="View_
        ↳ Results_Tree" enabled="true">
    <boolProp name="ResultCollector.error_logging">
        ↳ false</boolProp>
    <objProp>
        <name>saveConfig</name>
        <value class="SampleSaveConfiguration">
            <time>true</time>

```

```

    <latency>true</latency>
    <timestamp>true</timestamp>
    <success>true</success>
    <label>true</label>
    <code>true</code>
    <message>true</message>
    <threadName>true</threadName>
    <dataType>true</dataType>
    <encoding>>false</encoding>
    <assertions>true</assertions>
    <subresults>true</subresults>
    <responseData>>false</responseData>
    <samplerData>>false</samplerData>
    <xml>>false</xml>
    <fieldNames>true</fieldNames>
    <responseHeaders>>false</responseHeaders>
    <requestHeaders>>false</requestHeaders>
    <responseDataOnError>>false</
      ↪ responseDataOnError>
    <saveAssertionResultsFailureMessage>true</
      ↪ saveAssertionResultsFailureMessage>
    <assertionsResultsToSave>0</
      ↪ assertionsResultsToSave>
    <bytes>true</bytes>
    <sentBytes>true</sentBytes>
    <url>true</url>
    <threadCounts>true</threadCounts>
    <idleTime>true</idleTime>
    <connectTime>true</connectTime>
  </value>
</objProp>
  <stringProp name="filename"></stringProp>
</ResultCollector>
<hashTree/>
</hashTree>
</hashTree>
</hashTree>
</hashTree>
</jmeterTestPlan>

```

Listing C.1: GetMean.jmx

C.2 GetAPI Test plan

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.5">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan"
      ↪ testname="Test_Plan" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">>false</
        ↪ boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">>true</
        ↪ boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">>false</
        ↪ boolProp>
      <elementProp name="TestPlan.user_defined_variables"
        ↪ elementType="Arguments" guiclass="ArgumentsPanel"
        ↪ testclass="Arguments" testname="User_Defined_
        ↪ Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></
        ↪ stringProp>
    </TestPlan>
  <hashTree>
    <Arguments guiclass="ArgumentsPanel" testclass="
      ↪ Arguments" testname="User_Defined_Variables"
      ↪ enabled="true">
      <collectionProp name="Arguments.arguments">
        <elementProp name="threads" elementType="Argument">
          <stringProp name="Argument.name">threads</
            ↪ stringProp>
          <stringProp name="Argument.value">${_P(threads)
            ↪ }</stringProp>
          <stringProp name="Argument.metadata">=</stringProp
            ↪ >
          <stringProp name="Argument.desc">${_P(threads)}</
            ↪ stringProp>
        </elementProp>
        <elementProp name="rampup" elementType="Argument">
```

```

    <stringProp name="Argument.name">rampup</
      ↪ stringProp>
    <stringProp name="Argument.value">${_P(rampup)}</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</stringProp
      ↪ >
    <stringProp name="Argument.desc">${_P(rampup)}</
      ↪ stringProp>
  </elementProp>
  <elementProp name="loops" elementType="Argument">
    <stringProp name="Argument.name">loops</stringProp
      ↪ >
    <stringProp name="Argument.value">${_P(loops)}</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</stringProp
      ↪ >
    <stringProp name="Argument.desc">${_P(loops)}</
      ↪ stringProp>
  </elementProp>
</collectionProp>
</Arguments>
<hashTree/>
<ThreadGroup guiclass="ThreadGroupGui" testclass="
  ↪ ThreadGroup" testname="ThreadGroup" enabled="true
  ↪ ">
  <stringProp name="ThreadGroup.on_sample_error">
    ↪ continue</stringProp>
  <elementProp name="ThreadGroup.main_controller"
    ↪ elementType="LoopController" guiclass="
    ↪ LoopControlPanel" testclass="LoopController"
    ↪ testname="LoopController" enabled="true">
    <boolProp name="LoopController.continue_forever">
      ↪ false</boolProp>
    <stringProp name="LoopController.loops">${loops}</
      ↪ stringProp>
  </elementProp>
  <stringProp name="ThreadGroup.num_threads">${threads
    ↪ }</stringProp>
  <stringProp name="ThreadGroup.ramp_time">${rumpup}</
    ↪ stringProp>

```

```

<boolProp name="ThreadGroup.scheduler">false</boolProp>
  ↪ >
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<boolProp name="ThreadGroup.
  ↪ same_user_on_next_iteration">true</boolProp>
</ThreadGroup>
<hashTree>
  <CounterConfig guiclass="CounterConfigGui" testclass="
    ↪ CounterConfig" testname="Counter" enabled="false
    ↪ ">
  <stringProp name="CounterConfig.start">100</
    ↪ stringProp>
  <stringProp name="CounterConfig.end">1000</
    ↪ stringProp>
  <stringProp name="CounterConfig.incr">1</stringProp>
  <stringProp name="CounterConfig.name">ID</stringProp
    ↪ >
  <stringProp name="CounterConfig.format"></stringProp
    ↪ >
  <boolProp name="CounterConfig.per_user">false</
    ↪ boolProp>
</CounterConfig>
<hashTree/>
<RandomVariableConfig guiclass="TestBeanGUI" testclass
  ↪ ="RandomVariableConfig" testname="Random_
  ↪ Variable" enabled="true">
  <stringProp name="maximumValue">1000</stringProp>
  <stringProp name="minimumValue">100</stringProp>
  <stringProp name="outputFormat"></stringProp>
  <boolProp name="perThread">false</boolProp>
  <stringProp name="randomSeed"></stringProp>
  <stringProp name="variableName">ID</stringProp>
</RandomVariableConfig>
<hashTree/>
<HTTPSamplerProxy guiclass="HttpTestSampleGui"
  ↪ testclass="HTTPSamplerProxy" testname="HTTP_
  ↪ Request" enabled="true">
  <elementProp name="HTTPsampler.Arguments"
    ↪ elementType="Arguments" guiclass="
    ↪ HTTPArgumentsPanel" testclass="Arguments"

```

```

↪ testname="User_Defined_Variables" enabled="
↪ true">
<collectionProp name="Arguments.arguments">
  <elementProp name="id" elementType="HTTPArgument
↪ ">
    <boolProp name="HTTPArgument.always_encode">
↪ false</boolProp>
    <stringProp name="Argument.value">${ID}</
↪ stringProp>
    <stringProp name="Argument.metadata">=</
↪ stringProp>
    <boolProp name="HTTPArgument.use_equals">>true
↪ </boolProp>
    <stringProp name="Argument.name">id</
↪ stringProp>
  </elementProp>
</collectionProp>
</elementProp>
<stringProp name="HTTPSampler.domain">getwebapi.
↪ azurewebsites.net</stringProp>
<stringProp name="HTTPSampler.port">443</stringProp>
<stringProp name="HTTPSampler.protocol">https</
↪ stringProp>
<stringProp name="HTTPSampler.contentEncoding"></
↪ stringProp>
<stringProp name="HTTPSampler.path">/api/GetWebAPI</
↪ stringProp>
<stringProp name="HTTPSampler.method">GET</
↪ stringProp>
<boolProp name="HTTPSampler.follow_redirects">>true</
↪ boolProp>
<boolProp name="HTTPSampler.auto_redirects">>false</
↪ boolProp>
<boolProp name="HTTPSampler.use_keepalive">>true</
↪ boolProp>
<boolProp name="HTTPSampler.DO_MULTIPART_POST">>false
↪ </boolProp>
<stringProp name="HTTPSampler.embedded_url_re"></
↪ stringProp>
<stringProp name="HTTPSampler.connect_timeout"></
↪ stringProp>

```

```

    <stringProp name="HTTPSampler.response_timeout"></
      ↪ stringProp>
  </HTTPSamplerProxy>
<hashTree>
  <ResultCollector guiclass="SummaryReport" testclass=
    ↪ "ResultCollector" testname="Summary_Report"
    ↪ enabled="true">
  <boolProp name="ResultCollector.error_logging">
    ↪ false</boolProp>
  <objProp>
    <name>saveConfig</name>
    <value class="SampleSaveConfiguration">
      <time>true</time>
      <latency>true</latency>
      <timestamp>true</timestamp>
      <success>true</success>
      <label>true</label>
      <code>true</code>
      <message>true</message>
      <threadName>true</threadName>
      <dataType>true</dataType>
      <encoding>false</encoding>
      <assertions>true</assertions>
      <subresults>true</subresults>
      <responseData>false</responseData>
      <samplerData>false</samplerData>
      <xml>false</xml>
      <fieldNames>true</fieldNames>
      <responseHeaders>false</responseHeaders>
      <requestHeaders>false</requestHeaders>
      <responseDataOnError>false</
        ↪ responseDataOnError>
      <saveAssertionResultsFailureMessage>true</
        ↪ saveAssertionResultsFailureMessage>
      <assertionsResultsToSave>0</
        ↪ assertionsResultsToSave>
      <bytes>true</bytes>
      <sentBytes>true</sentBytes>
      <url>true</url>
      <threadCounts>true</threadCounts>
      <idleTime>true</idleTime>

```

```

        <connectTime>true</connectTime>
    </value>
</objProp>
    <stringProp name="filename"></stringProp>
</ResultCollector>
<hashTree/>
<ResultCollector guiclass="ViewResultsFullVisualizer
    ↪ " testclass="ResultCollector" testname="View_
    ↪ Results_Tree" enabled="true">
    <boolProp name="ResultCollector.error_logging">
        ↪ true</boolProp>
    <objProp>
        <name>saveConfig</name>
        <value class="SampleSaveConfiguration">
            <time>true</time>
            <latency>true</latency>
            <timestamp>true</timestamp>
            <success>true</success>
            <label>true</label>
            <code>true</code>
            <message>true</message>
            <threadName>true</threadName>
            <dataType>true</dataType>
            <encoding>>false</encoding>
            <assertions>true</assertions>
            <subresults>true</subresults>
            <responseData>>false</responseData>
            <samplerData>>false</samplerData>
            <xml>>false</xml>
            <fieldNames>true</fieldNames>
            <responseHeaders>>false</responseHeaders>
            <requestHeaders>>false</requestHeaders>
            <responseDataOnError>>false</
                ↪ responseDataOnError>
            <saveAssertionResultsFailureMessage>true</
                ↪ saveAssertionResultsFailureMessage>
            <assertionsResultsToSave>0</
                ↪ assertionsResultsToSave>
            <bytes>true</bytes>
            <sentBytes>true</sentBytes>
            <url>true</url>

```



```

        <threadCounts>true</threadCounts>
        <idleTime>true</idleTime>
        <connectTime>true</connectTime>
    </value>
</objProp>
    <stringProp name="filename"></stringProp>
</ResultCollector>
    <hashTree/>
</hashTree>
</hashTree>
</hashTree>
</hashTree>
</jmeterTestPlan>

```

Listing C.2: GetAPI.jmx

C.3 PostAPI Test plan

```

<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.5">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan"
      ↪ testname="Test_Plan" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</
        ↪ boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">true</
        ↪ boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">false</
        ↪ boolProp>
      <elementProp name="TestPlan.user_defined_variables"
        ↪ elementType="Arguments" guiclass="ArgumentsPanel"
        ↪ testclass="Arguments" testname="User_Defined_
        ↪ Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></
        ↪ stringProp>
    </TestPlan>
  </hashTree>
</jmeterTestPlan>

```

```

<hashTree>
  <Arguments guiclass="ArgumentsPanel" testclass="
    ↪ Arguments" testname="User_Defined_Variables"
    ↪ enabled="true">
    <collectionProp name="Arguments.arguments">
      <elementProp name="threads" elementType="Argument">
        <stringProp name="Argument.name">threads</
          ↪ stringProp>
        <stringProp name="Argument.value">${_P(threads)}
          ↪ }</stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(threads)}</
          ↪ stringProp>
      </elementProp>
      <elementProp name="rampup" elementType="Argument">
        <stringProp name="Argument.name">rampup</
          ↪ stringProp>
        <stringProp name="Argument.value">${_P(rampup)}</
          ↪ stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(rampup)}</
          ↪ stringProp>
      </elementProp>
      <elementProp name="loops" elementType="Argument">
        <stringProp name="Argument.name">loops</stringProp
          ↪ >
        <stringProp name="Argument.value">${_P(loops)}</
          ↪ stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(loops)}</
          ↪ stringProp>
      </elementProp>
    </collectionProp>
  </Arguments>
</hashTree/>
<ThreadGroup guiclass="ThreadGroupGui" testclass="
  ↪ ThreadGroup" testname="Thread_Group" enabled="true
  ↪ ">

```

```

<stringProp name="ThreadGroup.on_sample_error">
    ↪ continue</stringProp>
<elementProp name="ThreadGroup.main_controller"
    ↪ elementType="LoopController" guiclass="
    ↪ LoopControlPanel" testclass="LoopController"
    ↪ testname="LoopController" enabled="true">
<boolProp name="LoopController.continue_forever">
    ↪ false</boolProp>
<stringProp name="LoopController.loops">${loops}</
    ↪ stringProp>
</elementProp>
<stringProp name="ThreadGroup.num_threads">${threads
    ↪ }</stringProp>
<stringProp name="ThreadGroup.ramp_time">${rumpup}</
    ↪ stringProp>
<boolProp name="ThreadGroup.scheduler">>false</boolProp>
    ↪ >
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<boolProp name="ThreadGroup.
    ↪ same_user_on_next_iteration">>true</boolProp>
</ThreadGroup>
<hashTree>
<HTTPSamplerProxy guiclass="HttpTestSampleGui"
    ↪ testclass="HTTPSamplerProxy" testname="HTTP
    ↪ Request" enabled="true">
<boolProp name="HTTPSampler.postBodyRaw">>true</
    ↪ boolProp>
<elementProp name="HTTPsampler.Arguments"
    ↪ elementType="Arguments">
<collectionProp name="Arguments.arguments">
    <elementProp name="" elementType="HTTPArgument">
        <boolProp name="HTTPArgument.always_encode">
            ↪ false</boolProp>
        <stringProp name="Argument.value">{&#xd;
&quot;id&quot;;: &quot;${_UID()}&quot;;,&#xd;
&quot;categoryId&quot;;: &quot;3E4CEACD-D007-46EB-82D7-31
    ↪ F6141752B2&quot;;,&#xd;
&quot;categoryName&quot;;: &quot;Components, Road Frames&
    ↪ quot;;,&#xd;
&quot;sku&quot;;: &quot;FR-R38R-48&quot;;,&#xd;

```

```

    &quot;name&quot;;: &quot;LL Road Frame - Red, 48&quot;;,&#xd;
        ↪ ;
    &quot;description&quot;;: &quot;The product called \&quot;
        ↪ LL Road Frame - Red, 48\&quot;;&quot;;,&#xd;
    &quot;price&quot;;: 337.22,&#xd;
    &quot;tags&quot;;: [&#xd;
        {&#xd;
            &quot;id&quot;;: &quot;033D3826-2851-4B97-9464-59
                ↪ D3675175D4&quot;;,&#xd;
            &quot;name&quot;;: &quot;Tag-108&quot;;&#xd;
        },&#xd;
        {&#xd;
            &quot;id&quot;;: &quot;A2176C7A-4E0D-4283-AFAA-319
                ↪ A77E9C122&quot;;,&#xd;
            &quot;name&quot;;: &quot;Tag-19&quot;;&#xd;
        }&#xd;
    ]&#xd;
}</stringProp>
    <stringProp name="Argument.metadata">=</
        ↪ stringProp>
    </elementProp>
</collectionProp>
</elementProp>
<stringProp name="HTTPSampler.domain">fidelapp.
    ↪ azurewebsites.net</stringProp>
<stringProp name="HTTPSampler.port">443</stringProp>
<stringProp name="HTTPSampler.protocol">https</
    ↪ stringProp>
<stringProp name="HTTPSampler.contentEncoding"></
    ↪ stringProp>
<stringProp name="HTTPSampler.path">/api/FiDelApp</
    ↪ stringProp>
<stringProp name="HTTPSampler.method">POST</
    ↪ stringProp>
<boolProp name="HTTPSampler.follow_redirects">>true</
    ↪ boolProp>
<boolProp name="HTTPSampler.auto_redirects">>false</
    ↪ boolProp>
<boolProp name="HTTPSampler.use_keepalive">>true</
    ↪ boolProp>

```

```

<boolProp name="HTTPSampler.DO_MULTIPART_POST">>false
  ↪ </boolProp>
<stringProp name="HTTPSampler.embedded_url_re"></
  ↪ stringProp>
<stringProp name="HTTPSampler.connect_timeout"></
  ↪ stringProp>
<stringProp name="HTTPSampler.response_timeout"></
  ↪ stringProp>
</HTTPSamplerProxy>
<hashCode>
  <ResultCollector guiclass="SummaryReport" testclass=
    ↪ "ResultCollector" testname="Summary_Report"
    ↪ enabled="true">
  <boolProp name="ResultCollector.error_logging">
    ↪ false</boolProp>
  <objProp>
    <name>saveConfig</name>
    <value class="SampleSaveConfiguration">
      <time>true</time>
      <latency>true</latency>
      <timestamp>true</timestamp>
      <success>true</success>
      <label>false</label>
      <code>true</code>
      <message>true</message>
      <threadName>false</threadName>
      <dataType>false</dataType>
      <encoding>false</encoding>
      <assertions>false</assertions>
      <subresults>false</subresults>
      <responseData>false</responseData>
      <samplerData>true</samplerData>
      <xml>true</xml>
      <fieldNames>false</fieldNames>
      <responseHeaders>true</responseHeaders>
      <requestHeaders>true</requestHeaders>
      <responseDataOnError>false</
        ↪ responseDataOnError>
      <saveAssertionResultsFailureMessage>true</
        ↪ saveAssertionResultsFailureMessage>

```

```

        <assertionsResultsToSave>0</
            ↪ assertionsResultsToSave>
        <sentBytes>>true</sentBytes>
    </value>
</objProp>
    <stringProp name="filename"></stringProp>
</ResultCollector>
<hashTree/>
<ResultCollector guiclass="ViewResultsFullVisualizer
    ↪ " testclass="ResultCollector" testname="View_
    ↪ Results_Tree" enabled="true">
    <boolProp name="ResultCollector.error_logging">
        ↪ false</boolProp>
    <objProp>
        <name>saveConfig</name>
        <value class="SampleSaveConfiguration">
            <time>true</time>
            <latency>true</latency>
            <timestamp>true</timestamp>
            <success>true</success>
            <label>true</label>
            <code>true</code>
            <message>true</message>
            <threadName>true</threadName>
            <dataType>true</dataType>
            <encoding>>false</encoding>
            <assertions>true</assertions>
            <subresults>true</subresults>
            <responseData>>false</responseData>
            <samplerData>>false</samplerData>
            <xml>>false</xml>
            <fieldNames>true</fieldNames>
            <responseHeaders>>false</responseHeaders>
            <requestHeaders>>false</requestHeaders>
            <responseDataOnError>>false</
                ↪ responseDataOnError>
            <saveAssertionResultsFailureMessage>true</
                ↪ saveAssertionResultsFailureMessage>
            <assertionsResultsToSave>0</
                ↪ assertionsResultsToSave>
            <bytes>true</bytes>

```

```

        <sentBytes>true</sentBytes>
        <url>true</url>
        <threadCounts>true</threadCounts>
        <idleTime>true</idleTime>
        <connectTime>true</connectTime>
    </value>
</objProp>
    <stringProp name="filename"></stringProp>
</ResultCollector>
<hashTree/>
</hashTree>
</hashTree>
</hashTree>
</hashTree>
</jmeterTestPlan>

```

Listing C.3: PostAPI.jmx

C.4 CalculatePrimeNumbers Test plan

```

<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.5">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan"
      ↪ testname="Test_Plan" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">>false</
        ↪ boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">>true</
        ↪ boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">>false</
        ↪ boolProp>
      <elementProp name="TestPlan.user_defined_variables"
        ↪ elementType="Arguments" guiclass="ArgumentsPanel"
        ↪ testclass="Arguments" testname="User_Defined_
        ↪ Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
    </TestPlan>
  </hashTree>
</jmeterTestPlan>

```

```

    <stringProp name="TestPlan.user_define_classpath"></
      ↪ stringProp>
</TestPlan>
<hashTree>
  <Arguments guiclass="ArgumentsPanel" testclass="
    ↪ Arguments" testname="User_Defined_Variables"
    ↪ enabled="true">
    <collectionProp name="Arguments.arguments">
      <elementProp name="threads" elementType="Argument">
        <stringProp name="Argument.name">threads</
          ↪ stringProp>
        <stringProp name="Argument.value">${_P(threads)}
          ↪ </stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(threads)}</
          ↪ stringProp>
      </elementProp>
      <elementProp name="rampup" elementType="Argument">
        <stringProp name="Argument.name">rampup</
          ↪ stringProp>
        <stringProp name="Argument.value">${_P(rampup)}</
          ↪ stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(rampup)}</
          ↪ stringProp>
      </elementProp>
      <elementProp name="loops" elementType="Argument">
        <stringProp name="Argument.name">loops</stringProp
          ↪ >
        <stringProp name="Argument.value">${_P(loops)}</
          ↪ stringProp>
        <stringProp name="Argument.metadata">=</stringProp
          ↪ >
        <stringProp name="Argument.desc">${_P(loops)}</
          ↪ stringProp>
      </elementProp>
    </collectionProp>
  </Arguments>
</hashTree/>

```



```

<ThreadGroup guiclass="ThreadGroupGui" testclass="
  ↳ ThreadGroup" testname="Thread_Group" enabled="true
  ↳ ">
<stringProp name="ThreadGroup.on_sample_error">
  ↳ continue</stringProp>
<elementProp name="ThreadGroup.main_controller"
  ↳ elementType="LoopController" guiclass="
  ↳ LoopControlPanel" testclass="LoopController"
  ↳ testname="Loop_Controller" enabled="true">
<boolProp name="LoopController.continue_forever">
  ↳ false</boolProp>
<stringProp name="LoopController.loops">${loops}</
  ↳ stringProp>
</elementProp>
<stringProp name="ThreadGroup.num_threads">${threads
  ↳ }</stringProp>
<stringProp name="ThreadGroup.ramp_time">${rumpup}</
  ↳ stringProp>
<boolProp name="ThreadGroup.scheduler">>false</boolProp>
  ↳ >
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<boolProp name="ThreadGroup.
  ↳ same_user_on_next_iteration">>true</boolProp>
</ThreadGroup>
<hashTree>
  <RandomVariableConfig guiclass="TestBeanGUI" testclass
    ↳ ="RandomVariableConfig" testname="Random_
    ↳ Variable" enabled="true">
  <stringProp name="maximumValue">1000001</stringProp>
  <stringProp name="minimumValue">1000000</stringProp>
  <stringProp name="outputFormat"></stringProp>
  <boolProp name="perThread">>false</boolProp>
  <stringProp name="randomSeed"></stringProp>
  <stringProp name="variableName">NUMBER</stringProp>
  </RandomVariableConfig>
<hashTree/>
<HTTPSamplerProxy guiclass="HttpTestSampleGui"
  ↳ testclass="HTTPSamplerProxy" testname="HTTP_
  ↳ Request" enabled="true">

```

```

<elementProp name="HTTPSampler.Arguments"
  ↪ elementType="Arguments" guiclass="
  ↪ HTTPArgumentsPanel" testclass="Arguments"
  ↪ testname="User_Defined_Variables" enabled="
  ↪ true">
<collectionProp name="Arguments.arguments">
  <elementProp name="number" elementType="
    ↪ HTTPArgument">
    <boolProp name="HTTPArgument.always_encode">
      ↪ false</boolProp>
    <stringProp name="Argument.value">1000000</
      ↪ stringProp>
    <stringProp name="Argument.metadata">=</
      ↪ stringProp>
    <boolProp name="HTTPArgument.use_equals">>true
      ↪ </boolProp>
    <stringProp name="Argument.name">number</
      ↪ stringProp>
  </elementProp>
  <elementProp name="code" elementType="
    ↪ HTTPArgument">
    <boolProp name="HTTPArgument.always_encode">
      ↪ false</boolProp>
    <stringProp name="Argument.value">
      ↪ eGey3pmSHrb3HvoFf0gzMIIdLP7hthzrTLqwKvLwneQm7AzFuDC
      ↪ ==</stringProp>
    <stringProp name="Argument.metadata">=</
      ↪ stringProp>
    <boolProp name="HTTPArgument.use_equals">>true
      ↪ </boolProp>
    <stringProp name="Argument.name">code</
      ↪ stringProp>
  </elementProp>
</collectionProp>
</elementProp>
<stringProp name="HTTPSampler.domain">
  ↪ validationtestfunctions.azurewebsites.net</
  ↪ stringProp>
<stringProp name="HTTPSampler.port">443</stringProp>
<stringProp name="HTTPSampler.protocol">https</
  ↪ stringProp>

```

```

<stringProp name="HTTPSampler.contentEncoding"></
  ↳ stringProp>
<stringProp name="HTTPSampler.path">api/
  ↳ CalculatePrimeNumbers</stringProp>
<stringProp name="HTTPSampler.method">GET</
  ↳ stringProp>
<boolProp name="HTTPSampler.follow_redirects">true</
  ↳ boolProp>
<boolProp name="HTTPSampler.auto_redirects">false</
  ↳ boolProp>
<boolProp name="HTTPSampler.use_keepalive">true</
  ↳ boolProp>
<boolProp name="HTTPSampler.DO_MULTIPART_POST">false
  ↳ </boolProp>
<stringProp name="HTTPSampler.embedded_url_re"></
  ↳ stringProp>
<stringProp name="HTTPSampler.connect_timeout"></
  ↳ stringProp>
<stringProp name="HTTPSampler.response_timeout"></
  ↳ stringProp>
</HTTPSamplerProxy>
<hashTree>
  <ResultCollector guiclass="SummaryReport" testclass=
    ↳ "ResultCollector" testname="Summary_Report"
    ↳ enabled="true">
  <boolProp name="ResultCollector.error_logging">
    ↳ false</boolProp>
  <objProp>
    <name>saveConfig</name>
    <value class="SampleSaveConfiguration">
      <time>true</time>
      <latency>true</latency>
      <timestamp>true</timestamp>
      <success>true</success>
      <label>true</label>
      <code>true</code>
      <message>true</message>
      <threadName>true</threadName>
      <dataType>true</dataType>
      <encoding>false</encoding>
      <assertions>true</assertions>

```

```

    <subresults>true</subresults>
    <responseData>>false</responseData>
    <samplerData>>false</samplerData>
    <xml>>false</xml>
    <fieldNames>true</fieldNames>
    <responseHeaders>>false</responseHeaders>
    <requestHeaders>>false</requestHeaders>
    <responseDataOnError>>false</
      ↪ responseDataOnError>
    <saveAssertionResultsFailureMessage>true</
      ↪ saveAssertionResultsFailureMessage>
    <assertionsResultsToSave>0</
      ↪ assertionsResultsToSave>
    <bytes>true</bytes>
    <sentBytes>true</sentBytes>
    <threadCounts>true</threadCounts>
    <idleTime>true</idleTime>
    <connectTime>true</connectTime>
  </value>
</objProp>
<stringProp name="filename">C:\Users\a.penco\
  ↪ OneDrive - Reply\Desktop\summary.jtl</
  ↪ stringProp>
</ResultCollector>
<hashTree/>
<ResultCollector guiclass="ViewResultsFullVisualizer
  ↪ " testclass="ResultCollector" testname="View_
  ↪ Results_Tree" enabled="true">
  <boolProp name="ResultCollector.error_logging">
    ↪ false</boolProp>
  <objProp>
    <name>saveConfig</name>
    <value class="SampleSaveConfiguration">
      <time>true</time>
      <latency>true</latency>
      <timestamp>true</timestamp>
      <success>true</success>
      <label>true</label>
      <code>true</code>
      <message>true</message>
      <threadName>true</threadName>

```

```

<dataType>true</dataType>
<encoding>>false</encoding>
<assertions>true</assertions>
<subresults>true</subresults>
<responseData>>false</responseData>
<samplerData>>false</samplerData>
<xml>>false</xml>
<fieldNames>true</fieldNames>
<responseHeaders>>false</responseHeaders>
<requestHeaders>>false</requestHeaders>
<responseDataOnError>>false</
  ↳ responseDataOnError>
<saveAssertionResultsFailureMessage>true</
  ↳ saveAssertionResultsFailureMessage>
<assertionsResultsToSave>0</
  ↳ assertionsResultsToSave>
<bytes>true</bytes>
<sentBytes>true</sentBytes>
<url>true</url>
<threadCounts>true</threadCounts>
<idleTime>true</idleTime>
<connectTime>true</connectTime>
  </value>
</objProp>
  <stringProp name="filename"></stringProp>
</ResultCollector>
<hashTree/>
</hashTree>
</hashTree>
</hashTree>
</hashTree>
</jmeterTestPlan>

```

Listing C.4: CalculatePrimeNumbers.jmx

Appendix D

Prompt Engineering component

D.1 JMeter result files

JTL file

```
timeStamp , elapsed , label , responseCode , responseMessage , threadName ,  
  ↳ dataType , success , failureMessage , bytes , sentBytes , grpThreads ,  
  ↳ allThreads , URL , Latency , IdleTime , Connect  
timeStamp , elapsed , label , responseCode , responseMessage , threadName ,  
  ↳ dataType , success , failureMessage , bytes , sentBytes , grpThreads ,  
  ↳ allThreads , URL , Latency , IdleTime , Connect  
1699096653555 , 96198 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-65 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.  
  ↳ azurewebsites.net/api/GetMean?Country=Italy , 96193 , 0 , 96041  
1699096653279 , 96483 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-15 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.  
  ↳ azurewebsites.net/api/GetMean?Country=Italy , 96483 , 0 , 96307  
1699096653460 , 96303 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-668 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.  
  ↳ azurewebsites.net/api/GetMean?Country=Japan , 96303 , 0 , 96128  
1699096653791 , 95972 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-902 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.  
  ↳ azurewebsites.net/api/GetMean?Country=India , 95972 , 0 , 95795  
1699096653222 , 96541 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-538 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.  
  ↳ azurewebsites.net/api/GetMean?Country=Japan , 96541 , 0 , 96349  
1699096653235 , 96528 , HTTP Request , 429 , Too Many Requests , Thread  
  ↳ Group 1-646 , , false , , 158 , 155 , 1000 , 1000 , https://getmean.
```

```

    ↪ azurewebsites.net/api/GetMean?Country=Japan,96528,0,96351
1699096653714,96050,HTTP Request,429,Too Many Requests,Thread
    ↪ Group 1-970,,false,,158,155,1000,1000,https://getmean.
    ↪ azurewebsites.net/api/GetMean?Country=India,96050,0,95866
1699096653624,96140,HTTP Request,429,Too Many Requests,Thread
    ↪ Group 1-139,,false,,158,155,1000,1000,https://getmean.
    ↪ azurewebsites.net/api/GetMean?Country=Japan,96140,0,95950
1699096653425,96339,HTTP Request,429,Too Many Requests,Thread
    ↪ Group 1-885,,false,,158,155,1000,1000,https://getmean.
    ↪ azurewebsites.net/api/GetMean?Country=Italy,96339,0,96149
...
...
...
1699096898618,315,HTTP Request,200,OK,Thread Group 1-642,text,true
    ↪ ,,195,155,2,2,https://getmean.azurewebsites.net/api/GetMean?
    ↪ Country=India,315,0,0
1699096898700,325,HTTP Request,200,OK,Thread Group 1-851,text,true
    ↪ ,,195,155,2,2,https://getmean.azurewebsites.net/api/GetMean?
    ↪ Country=India,325,0,0
1699096898933,315,HTTP Request,200,OK,Thread Group 1-642,text,true
    ↪ ,,195,155,2,2,https://getmean.azurewebsites.net/api/GetMean?
    ↪ Country=India,315,0,0
1699096899025,323,HTTP Request,200,OK,Thread Group 1-851,text,true
    ↪ ,,198,155,1,1,https://getmean.azurewebsites.net/api/GetMean?
    ↪ Country=Japan,323,0,0

```

Listing D.1: GetMean.jtl

TXT summary file

```

Creating summariser <summary>
Created the tree successfully using D:\a\1\s\GetMean.jmx
Starting standalone test @ November 4, 2023 11:17:28 AM UTC
    ↪ (1699096648850)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump
    ↪ message on port 4445
summary + 27611 in 00:01:59 = 231.1/s Avg: 4154 Min: 19 Max:
    ↪ 102787 Err: 23309 (84.42%) Active: 1000 Started: 1000
    ↪ Finished: 0
summary + 50478 in 00:00:30 = 1683.0/s Avg: 553 Min: 18 Max:
    ↪ 8504 Err: 44202 (87.57%) Active: 796 Started: 1000 Finished
    ↪ : 204

```

```

summary = 78089 in 00:02:29 = 522.5/s Avg: 1826 Min: 18 Max:
↳ 102787 Err: 67511 (86.45%)
summary + 7010 in 00:00:30 = 233.7/s Avg: 3191 Min: 1064 Max:
↳ 11273 Err: 791 (11.28%) Active: 666 Started: 1000 Finished
↳ : 334
summary = 85099 in 00:02:59 = 474.2/s Avg: 1939 Min: 18 Max:
↳ 102787 Err: 68302 (80.26%)
summary + 6943 in 00:00:30 = 231.4/s Avg: 2559 Min: 530 Max:
↳ 7988 Err: 798 (11.49%) Active: 470 Started: 1000 Finished
↳ : 530
summary = 92042 in 00:03:29 = 439.4/s Avg: 1985 Min: 18 Max:
↳ 102787 Err: 69100 (75.07%)
summary + 7252 in 00:00:30 = 241.7/s Avg: 1395 Min: 307 Max:
↳ 9672 Err: 428 (5.90%) Active: 76 Started: 1000 Finished:
↳ 924
summary = 99294 in 00:03:59 = 414.7/s Avg: 1942 Min: 18 Max:
↳ 102787 Err: 69528 (70.02%)
summary + 706 in 00:00:09 = 75.6/s Avg: 325 Min: 308 Max:
↳ 746 Err: 0 (0.00%) Active: 0 Started: 1000 Finished:
↳ 1000
summary = 100000 in 00:04:09 = 401.9/s Avg: 1931 Min: 18 Max:
↳ 102787 Err: 69528 (69.53%)
Tidying up ... @ November 4, 2023 11:21:39 AM UTC
↳ (1699096899348)
... end of run

```

Listing D.2: summary.txt

D.2 Generated JMeter input for GPT model

```

Result from the JTL file from the JMeter test in which are saved
↳ the errors and the description:
---
66587 x 429 Too Many Requests
2941 x 500 Internal Server Error
---
Summary results:
---
summary + 27611 in 00:01:59 = 231.1/s Avg: 4154 Min: 19 Max:
↳ 102787 Err: 23309 (84.42%) Active: 1000 Started: 1000
↳ Finished: 0

```



```

summary + 50478 in 00:00:30 = 1683.0/s Avg: 553 Min: 18 Max:
↳ 8504 Err: 44202 (87.57%) Active: 796 Started: 1000 Finished
↳ : 204
summary = 78089 in 00:02:29 = 522.5/s Avg: 1826 Min: 18 Max:
↳ 102787 Err: 67511 (86.45%)
summary + 7010 in 00:00:30 = 233.7/s Avg: 3191 Min: 1064 Max:
↳ 11273 Err: 791 (11.28%) Active: 666 Started: 1000 Finished
↳ : 334
summary = 85099 in 00:02:59 = 474.2/s Avg: 1939 Min: 18 Max:
↳ 102787 Err: 68302 (80.26%)
summary + 6943 in 00:00:30 = 231.4/s Avg: 2559 Min: 530 Max:
↳ 7988 Err: 798 (11.49%) Active: 470 Started: 1000 Finished
↳ : 530
summary = 92042 in 00:03:29 = 439.4/s Avg: 1985 Min: 18 Max:
↳ 102787 Err: 69100 (75.07%)
summary + 7252 in 00:00:30 = 241.7/s Avg: 1395 Min: 307 Max:
↳ 9672 Err: 428 (5.90%) Active: 76 Started: 1000 Finished:
↳ 924
summary = 99294 in 00:03:59 = 414.7/s Avg: 1942 Min: 18 Max:
↳ 102787 Err: 69528 (70.02%)
summary + 706 in 00:00:09 = 75.6/s Avg: 325 Min: 308 Max:
↳ 746 Err: 0 (0.00%) Active: 0 Started: 1000 Finished:
↳ 1000
summary = 100000 in 00:04:09 = 401.9/s Avg: 1931 Min: 18 Max:
↳ 102787 Err: 69528 (69.53%)
---

```

Listing D.3: TestResults.txt

D.3 Input for GPT model

```

Provide improved implementation example taking into account the
↳ following:
----
JTL Results:
---
Result from the JTL file from the JMeter test in which are saved
↳ the errors and the description:
---
66587 x 429 Too Many Requests

```

2941 x 500 Internal Server Error

Summary results:

summary + 27611 in 00:01:59 = 231.1/s Avg: 4154 Min: 19 Max:

↪ 102787 Err: 23309 (84.42%) Active: 1000 Started: 1000

↪ Finished: 0

summary + 50478 in 00:00:30 = 1683.0/s Avg: 553 Min: 18 Max:

↪ 8504 Err: 44202 (87.57%) Active: 796 Started: 1000 Finished

↪ : 204

summary = 78089 in 00:02:29 = 522.5/s Avg: 1826 Min: 18 Max:

↪ 102787 Err: 67511 (86.45%)

summary + 7010 in 00:00:30 = 233.7/s Avg: 3191 Min: 1064 Max:

↪ 11273 Err: 791 (11.28%) Active: 666 Started: 1000 Finished

↪ : 334

summary = 85099 in 00:02:59 = 474.2/s Avg: 1939 Min: 18 Max:

↪ 102787 Err: 68302 (80.26%)

summary + 6943 in 00:00:30 = 231.4/s Avg: 2559 Min: 530 Max:

↪ 7988 Err: 798 (11.49%) Active: 470 Started: 1000 Finished

↪ : 530

summary = 92042 in 00:03:29 = 439.4/s Avg: 1985 Min: 18 Max:

↪ 102787 Err: 69100 (75.07%)

summary + 7252 in 00:00:30 = 241.7/s Avg: 1395 Min: 307 Max:

↪ 9672 Err: 428 (5.90%) Active: 76 Started: 1000 Finished:

↪ 924

summary = 99294 in 00:03:59 = 414.7/s Avg: 1942 Min: 18 Max:

↪ 102787 Err: 69528 (70.02%)

summary + 706 in 00:00:09 = 75.6/s Avg: 325 Min: 308 Max:

↪ 746 Err: 0 (0.00%) Active: 0 Started: 1000 Finished:

↪ 1000

summary = 100000 in 00:04:09 = 401.9/s Avg: 1931 Min: 18 Max:

↪ 102787 Err: 69528 (69.53%)

Function Code:

using System;

using System.IO;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Azure.WebJobs;

using Microsoft.Azure.WebJobs.Extensions.Http;

using Microsoft.AspNetCore.Http;

```

using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using System.Net.Http;
using Microsoft.Azure.Cosmos;
using System.ComponentModel;
using Azure;
using System.Collections.Generic;
using System.Collections.Concurrent;
using System.Linq;

namespace GetMean
{
    public static class GetMean
    {
        private static readonly string cosmosDbEndpoint =
            ↪ Environment.GetEnvironmentVariable("DBENDPOINT",
            ↪ EnvironmentVariableTarget.Process);
        private static readonly string cosmosDbKey = Environment.
            ↪ GetEnvironmentVariable("DBKEY",
            ↪ EnvironmentVariableTarget.Process);
        private static readonly string databaseId = Environment.
            ↪ GetEnvironmentVariable("DBID",
            ↪ EnvironmentVariableTarget.Process);
        private static readonly string containerId = Environment.
            ↪ GetEnvironmentVariable("CONTAINERID",
            ↪ EnvironmentVariableTarget.Process);
        private static readonly CosmosClient cosmosClient = new
            ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey);
        private static readonly Microsoft.Azure.Cosmos.Container
            ↪ cosmosContainer = cosmosClient.GetContainer(
            ↪ databaseId, containerId);

        [FunctionName("GetMean")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get",
            ↪ Route = null)] HttpRequest req,
            ILogger log)
        {
            try
            {
                string country = req.Query["Country"];

                // Validate that the country is provided

```

```

if (string.IsNullOrEmpty(country))
{
    return new BadRequestObjectResult("The '
        ↪ Country' query parameter is required.");
}

FeedIterator<Volcano> queryResultSetIterator =
    ↪ cosmosContainer.GetItemQueryIterator<Volcano>
    ↪ >();
double sum = 0;
int count = 0;

while (queryResultSetIterator.HasMoreResults)
{
    FeedResponse<Volcano> currentResultSet = await
        ↪ queryResultSetIterator.ReadNextAsync();

    foreach (Volcano volcano in currentResultSet)
    {
        if (volcano.Country == country)
        {
            if (int.TryParse(volcano.Elevation,
                ↪ out int parsedElevation))
            {
                sum += parsedElevation;
                count++;
            }
            else
            {
                log.LogWarning($"Could not parse
                    ↪ elevation '{volcano.Elevation
                    ↪ }' for volcano '{volcano.
                    ↪ VolcanoName}'. This elevation
                    ↪ will be ignored.");
            }
        }
    }
}

// If no volcanoes found, return not found
if (count == 0)
{
    return new NotFoundResult();
}

```

```

    }

    // Calculate the mean elevation
    double meanElevation = Math.Round(sum / count, 3);

    return new OkObjectResult(meanElevation);
}
catch (Exception ex)
{
    log.LogError(ex, "An error occurred while
        ↪ retrieving data from Cosmos DB.");
    return new StatusCodeResult(StatusCodes.
        ↪ Status500InternalServerError);
}
}

}

public class Volcano
{
    public string VolcanoName { get; set; }
    public string Country { get; set; }
    public string Region { get; set; }
    public Location Location { get; set; }
    public string Elevation { get; set; }
    public string Type { get; set; }
    public string Status { get; set; }
    public string LastKnownEruption { get; set; }
    public string Id { get; set; }
}

public class Location
{
    public string Type { get; set; }
    public List<double> Coordinates { get; set; }
}
}
---
----
```

Listing D.4: InputForGPT.txt

Appendix E

Test with different program without prompt engineering

E.1 Test results with GPT-3

Prototype response for GetMean

The JTL file contains results from a JMeter test. It includes information such as timeStamp, elapsed time, label, response code, response message, thread name, data type, success, failure message, bytes, sent bytes, group threads, all threads, URL, latency, idle time, and connection.

The function code is written in C# and is a Azure Function named "*Get-Mean*". It is triggered by an HTTP GET request and retrieves data from a Cosmos DB. The function calculates the mean elevation of volcanoes in a specific country. It reads the "*Country*" query parameter from the request and queries the Cosmos DB container for volcanoes in that country. It then calculates the mean elevation by summing up the elevations and dividing by the number of volcanoes found. The result is returned as a HTTP response.

The function uses the Cosmos Client SDK to connect to the Cosmos DB using the endpoint and key provided as environment variables. It retrieves the Cosmos DB container and iterates over the query results to calculate the mean elevation.

The function handles various scenarios, such as when the "*Country*" query

parameter is not provided or when no volcanoes are found for the given country. It logs any errors that occur during the execution and returns appropriate HTTP responses.

Prototype response for GetAPI

The JTL results show the details of the JMeter test, including the timestamp, elapsed time, label, response code, response message, thread name, data type, success status, failure message, bytes, sent bytes, group threads, all threads, URL, latency, idle time, and connect time.

The function code is a C# program that uses Azure Functions to create a HTTP trigger function named "GetAPI". This function retrieves data from a Cosmos DB container using the provided ID as a query parameter. If the ID is not provided, a `BadRequestObjectResult` is returned. If the item is not found in the Cosmos DB, a `NotFoundResult` is returned. If any other exception occurs during the retrieval process, a `StatusCodeResult` with a status code of 500 is returned. The `RequestDocument` class represents the structure of the item in the Cosmos DB container, with properties for the ID, category ID, category name, SKU, name, description, price, and a list of tags. The `Tag` class represents a tag with an ID and name.

Prototype response for PostAPI

From the JTL results, it looks like there were a total of 100 HTTP requests made, with all of them resulting in an internal server error (500). The average response time for the requests was 923 milliseconds, with the minimum being 154 milliseconds and the maximum being 6885 milliseconds.

The function code provided is a C# Azure Function with an HTTP trigger. It reads a JSON request body, deserializes it into a `RequestDocument` object, and then inserts the document into a Cosmos DB container using the Cosmos DB .NET SDK.

There is also a `RequestDocument` class defined, with properties for the document fields, such as `id`, `categoryId`, `categoryName`, `sku`, `name`, `description`, `price`, and `tags`.

It appears that there may be an issue with the Azure Function, as all the HTTP requests resulted in an internal server error. Further debugging and investigation would be needed to determine the cause of the error.

E.2 Program test results with GPT4

Prototype response for GetMean

The above code is an Azure function in C# that retrieves the mean (average) elevation of volcanoes in a specific country. This function retrieves data stored in a Cosmos DB, calculates the mean elevation and returns it.

The Azure function is named "*GetMean*" and it is an HTTP trigger type of function, meaning it is invoked when an HTTP request is made. It accepts anonymous authorization with only the GET method.

The function retrieves the country as a parameter from the HTTP request, makes a query to Cosmos DB to retrieve an iterator over all items (representing volcanoes), then it aggregates the total elevation of all volcanoes found in the specified country, keeping a count of total found items as well.

Once the iteration is done, it calculates the average (mean) by dividing the total elevation by the number of volcanoes found. If no volcanoes are found, it returns a "not found" result. It rounds the mean to three decimal places before returning it.

When the elevation of a volcano can't be parsed as an integer, a warning message is logged and the elevation is ignored in the sum calculation.

The class `Volcano` and nested class `Location` provide the schema for the data expected from Cosmos DB.

If an exception occurs during retrieval or processing, the function logs the error and returns the HTTP status 500 (internal server error). This error does not leak implementation details to clients and maintains server security. Still, it allows debugging and error tracing using logs.

Environmental variables store Cosmos DB's endpoint, key, and specified database and container IDs. Setting sensitive data in environment variables is a common security best practice.

Prototype response for GetAPI

This report presents the results from a JMeter test where HTTP requests have been sent to the *GetAPI* Azure function. The Azure function retrieves particular data from a Cosmos DB database.

However, the test results demonstrate that the function returned the HTTP

status code 404 (Not Found) for all the HTTP requests. This status code implies that the requested resource could not be found on the server and could indicate either a missing function or missing data in the database.

A few request details from the JTL file:

- The first request (<https://getwebapi.azurewebsites.net/api/GetWebAPI?id=168>) took 979 milliseconds to complete and failed with a 404 status code.
- The second request (<https://getwebapi.azurewebsites.net/api/GetWebAPI?id=476>) took 886 milliseconds to return a 404 status code.
- Other requests reveal similar failure patterns.

Overall, 65 out of 100 HTTP requests failed, resulting in an error rate of 65%. The Azure function code reveals that each HTTP request must provide an "id" as a query parameter. The function uses this "id" to read an item from the Cosmos DB database. If the "id" is not provided in the request or if the corresponding item is not found in the database, an error will be returned.

It is recommended to double-check if the "id" query parameters used in the test are valid and correspond to existing items in the Cosmos DB. Additionally, it might be useful to verify the configuration and availability of the Cosmos DB and the Azure function.

Prototype response for PostAPI

Based on the results from the JMeter test, all 100 HTTP requests to the Azure app <https://fidelapp.azurewebsites.net/api/FiDelApp> resulted in a 500 Internal Server Error. The test was performed by ten threads in just over 8 seconds and had an average response time of 400 milliseconds.

The error rate was 100%, indicating a severe issue with the request or the server. Notably, the test reported that the maximum response time was significantly high at 5375 milliseconds or about 5.4 seconds. This anomaly is worth investigating as it could suggest some potential issues or bottlenecks with the server performance.

The function code is implemented as a HTTP-triggered Azure function

in C#. It listens to HTTP POST requests, reads and deserializes the JSON body of the request into a **RequestDocument** object, performs some basic data validation, and then attempts to insert the received data into a Cosmos DB database.

The 500 Internal Server Error status means that the server encountered an unexpected condition that prevented it from fulfilling the request. The error is reported in the catch block that deals with exceptions that might have been thrown while inserting data into Cosmos DB. This leads to the suggestion that the exception may have roots in Cosmos DB manipulation, such as issues with the Cosmos DB settings or the structure of the inserted document.

Given the error message and code from the JTL file, one could start debugging by verifying whether the `cosmosDbEndpoint`, `cosmosDbKey`, `databaseId`, and `containerId` are correctly configured, the `cosmosContainer` exists and is reachable and ensuring the inserted document complies with the defined **RequestDocument** class in terms of structure and datatypes.

Appendix F

Prompt Test

F.1 Prompt Test using only system message

DevOptimizeAI response for GetMean

From the JTL Results, one can observe that your overall average response time is 3466 milliseconds which is quite high. To optimize your function, you may consider the following points:

1. **Database Calls:** By looking at your code it seems clear that the function makes a lot of sequential database calls when fetching and processing data from the Cosmos DB. Cosmos DB provides a lot of advanced query features that you could use to perform calculations like SUM or AVERAGE directly in the database. This would reduce your data transfer over the network and decrease the load on your function. Look into Refactoring the code to use stored procedures or Triggers inside Cosmos DB.
2. **Parallel Processing:** Some parts of your code do not depend on the results of previous iterations and can therefore be parallelized. For example, the processing of the Volcano objects in the loop can run concurrently using `Parallel.ForEach` or `Task.WhenAll`.
3. **Caching:** You should consider implementing caching. Depending on the use case and how often the data changes, you may cache data that are expensive to retrieve and does not change often, such as Volcanos elevation, avoiding a database hit on every request and thus improve

your function performance.

4. **Tracing:** You should also consider profiling your function to get more detailed insights on how it performs. Tools like Application Insights can help to identify bottlenecks in your code.
5. **Sharding:** Check also your Cosmos DB setup. If data is spread over multiple partitions, performance can be affected.

Finally, remember that improving performance often comes at the cost of increased complexity, so always profile and measure your changes to make sure that they are actually providing the expected benefits.

DevOptimizeAI response for GetAPI

Based on the JTL results from the JMeter test, there seems to be a recurring issue with HTTP requests returning a 404 error response code, which indicates that the requested resource could not be found on the server.

The reasons responsible for this response code might be:

- The item with the provided ID is not present in the database.
- The query parameter ID is not provided in the HTTP request.

This further leads to lower request throughput, longer average elapsed times, and an 86% error rate which is quite high.

In terms of the function code, it seems to be set up correctly to handle different cases and exceptions. It extracts the id parameter from the HTTP request as a string, checks whether it is present, and if it is, it tries to retrieve the corresponding item from Cosmos DB.

Nonetheless, based on the JMeter results, it seems like items might not be found in the DB consistently - thus the 404 error.

Here are some recommendations to better handle this situation:

- Implement more robust error handling and logging that includes specifics about the error, such as the specific ID that was not found.
- Check the database to ensure that data exists for the IDs that you are using in the tests.

- Validate the IDs being used in the test to ensure they are correctly formatted and of the correct type.
- Consider implementing a caching strategy to avoid hitting the DB for every request, which could help improve the response times and lessen the load on your database. But remember, caching needs to be managed carefully to avoid serving stale or inaccurate data.

These enhancements should help improve the response times, decrease the error rate, and increase system throughput.

DevOptimizeAI response for PostAPI

JMeter Test Result Analysis: The JMeter tests indicate that there is an issue with the web server returning HTTP 500 (Internal Server Error) status for all the requests, indeed a high error rate (20.00%) is reported. This usually indicates an unhandled exception in your code. In your context, from the HTTP requests, it seems the requests are being made to the `https://fidelapp.azurewebsites.net/api/FiDelApp` API endpoint, which is consistently failing.

Given the fact that all the requests resulted in a 500 - Internal Server Error, it may indicate the occurrence of exceptions or failures in application while trying to process these requests. In this case, the error logs would contain specific details as to what the exceptions or failures were, this information would be useful to focus the debugging efforts.

Code Review: The issue may come from the PostAPI Azure function.

In the code, the request body is deserialized and inserted into CosmosDB. Two cases are handled:

1. The case where the 'id' in the request is null or empty returns a 'BadRequest' status, which is not the case as per the test result.
2. The application returns HTTP status 500 on any exception while trying to insert data into Cosmos DB.

As JMeter doesn't test the logic inside your function, only its response, we can infer that the issue comes after the ID validation, in the interaction with Cosmos DB.

A good strategy would be to add more try-catch blocks around the smaller units of work (like the deserialization process, the CosmosDB submission).

This would allow you to return different HTTP status codes according to the error, which would give you more indications for your JMeter tests. You should also log meaningful error messages and/or exception details to accurately trace the root cause of issues. For critical issues, such as failure in connection to resources, it is recommended to implement advanced error handling strategies such as retries with exponential backoff.

Please rectify the issue by debugging HTTP 500 (Internal Server Error) status at <https://fidelapp.azurewebsites.net/api/FiDelApp> and performing tests again.

F.2 Prompt Test using only user message

DevOptimizeAI response for GetMean

Function Code:

```
1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Diagnostics;
15 using System.Collections.Generic;
16
17 namespace GetAPI
18 {
19     public static class GetAPI
20     {
21         private static readonly string cosmosDbEndpoint =
22             ↪ Environment.GetEnvironmentVariable("DBENDPOINT",
23             ↪ EnvironmentVariableTarget.Process);
24         private static readonly string cosmosDbKey = Environment.
25             ↪ GetEnvironmentVariable("DBKEY",
26             ↪ EnvironmentVariableTarget.Process);
27         private static readonly string databaseId = Environment.
```

```

24 ↪ GetEnvironmentVariable("DBID",
    ↪ EnvironmentVariableTarget.Process);
private static readonly string containerId = Environment.
25 ↪ GetEnvironmentVariable("CONTAINERID",
    ↪ EnvironmentVariableTarget.Process);
26
private static readonly CosmosClientOptions options =
    ↪ new CosmosClientOptions{ ApplicationName = "GetWebAPI"
    ↪ };
27 private static readonly CosmosClient cosmosClient = new
    ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey, options);
28 private static readonly Microsoft.Azure.Cosmos.
    ↪ Container cosmosContainer = cosmosClient.GetContainer(
    ↪ databaseId, containerId);
29
[FunctionName("GetWebAPI")]
30 public static async Task<IActionResult> Run(
31     [HttpTrigger(AuthorizationLevel.Anonymous, "get",
32     ↪ Route = null)] HttpRequest req,
33     ILogger log)
34 {
35     Stopwatch stopwatch = Stopwatch.StartNew();
36
37     try
38     {
39         string id = req.Query["id"];
40
41         if (string.IsNullOrEmpty(id))
42         {
43             ↪ return new BadRequestObjectResult("The 'id'
    ↪ query parameter is required.");
44         }
45
46         ItemResponse<RequestDocument> response = await
    ↪ cosmosContainer.ReadItemAsync<RequestDocument>(id, new
    ↪ PartitionKey(id));
47
48         stopwatch.Stop();
49
50         log.LogInformation($"Elapsed time: {stopwatch.
    ↪ Elapsed.Milliseconds} ms");
51         return new OkObjectResult(response.Resource);
52     }
53     catch (CosmosException ex) when (ex.StatusCode ==
    ↪ System.Net.HttpStatusCode.NotFound)
54     {
55         ↪ return new NotFoundResult();
56     }

```

```

57     catch (Exception ex)
58     {
59         stopwatch.Stop();
60         log.LogError(ex, $"An error occurred while
↪ retrieving data from Cosmos DB. Elapsed time: {
↪ stopwatch.Elapsed.Milliseconds} ms" );
61         return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
62     }
63 }
64 }
65
66 public class RequestDocument
67 {
68     public string id { get; set; }
69     public string categoryId { get; set; }
70     public string categoryName { get; set; }
71     public string sku { get; set; }
72     public string name { get; set; }
73     public string description { get; set; }
74     public double Price { get; set; }
75     public List<Tag> tags { get; set; }
76 }
77
78 public class Tag
79 {
80     public string id { get; set; }
81     public string name { get; set; }
82 }
83 }

```

This improved example has added a stopwatch to record the elapsed time of the database interaction. This can be useful for tracking and optimizing performance. Also, this version logs the time it takes for each call. This provides the ability to store instances where the function has been called, and how long those calls take. These log data can later be analyzed to identify system bottlenecks or areas that may require optimization. It also applies `CosmosClientOptions` with the name of the application, which could be useful for diagnostics and telemetry.

DevOptimizeAI response for GetAPI

Function Code:

```
1 using System;
```



```

2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Diagnostics;
15 using System.Collections.Generic;
16
17 namespace GetAPI
18 {
19     public static class GetAPI
20     {
21         private static readonly string cosmosDbEndpoint =
22             ↪ Environment.GetEnvironmentVariable("DBENDPOINT",
23             ↪ EnvironmentVariableTarget.Process);
24         private static readonly string cosmosDbKey = Environment.
25             ↪ GetEnvironmentVariable("DBKEY",
26             ↪ EnvironmentVariableTarget.Process);
27         private static readonly string databaseId = Environment.
28             ↪ GetEnvironmentVariable("DBID",
29             ↪ EnvironmentVariableTarget.Process);
30         private static readonly string containerId = Environment.
31             ↪ GetEnvironmentVariable("CONTAINERID",
32             ↪ EnvironmentVariableTarget.Process);
33
34         private static readonly CosmosClientOptions options =
35             ↪ new CosmosClientOptions{ ApplicationName = "GetWebAPI"
36             ↪ };
37         private static readonly CosmosClient cosmosClient = new
38             ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey, options);
39         private static readonly Microsoft.Azure.Cosmos.
40             ↪ Container cosmosContainer = cosmosClient.GetContainer(
41             ↪ databaseId, containerId);
42
43         [FunctionName("GetWebAPI")]
44         public static async Task<IActionResult> Run(
45             [HttpTrigger(AuthorizationLevel.Anonymous, "get",
46             ↪ Route = null)] HttpRequest req,
47             ILogger log)
48         {
49             Stopwatch stopwatch = Stopwatch.StartNew();

```

```

36
37     try
38     {
39         string id = req.Query["id"];
40
41         if (string.IsNullOrEmpty(id))
42         {
43             return new BadRequestObjectResult("The 'id'
↪ query parameter is required.");
44         }
45
46         ItemResponse<RequestDocument> response = await
↪ cosmosContainer.ReadItemAsync<RequestDocument>(id, new
↪ PartitionKey(id));
47
48         stopwatch.Stop();
49
50         log.LogInformation($"Elapsed time: {stopwatch.
↪ Elapsed.Milliseconds} ms");
51         return new OkObjectResult(response.Resource);
52     }
53     catch (CosmosException ex) when (ex.StatusCode ==
↪ System.Net.HttpStatusCode.NotFound)
54     {
55         return new NotFoundResult();
56     }
57     catch (Exception ex)
58     {
59         stopwatch.Stop();
60         log.LogError(ex, $"An error occurred while
↪ retrieving data from Cosmos DB. Elapsed time: {
↪ stopwatch.Elapsed.Milliseconds} ms" );
61         return new StatusCodeResult(StatusCode.
↪ Status500InternalServerError);
62     }
63 }
64 }
65
66 public class RequestDocument
67 {
68     public string id { get; set; }
69     public string categoryId { get; set; }
70     public string categoryName { get; set; }
71     public string sku { get; set; }
72     public string name { get; set; }
73     public string description { get; set; }
74     public double Price { get; set; }
75     public List<Tag> tags { get; set; }

```

```

76 }
77
78 public class Tag
79 {
80     public string id { get; set; }
81     public string name { get; set; }
82 }
83 }

```

This improved example has added a stopwatch to record the elapsed time of the database interaction. This can be useful for tracking and optimizing performance. Also, this version logs the time it takes for each call. This provides the ability to store instances where the function has been called, and how long those calls take. These log data can later be analyzed to identify system bottlenecks or areas that may require optimization. It also applies `CosmosClientOptions` with the name of the application, which could be useful for diagnostics and telemetry.

DevOptimizeAI response for PostAPI

Based on the error log, it is successful with the HTTP connection and with receiving the incoming HTTP request but it fails while performing an action on the Cosmos DB, during the Cosmos DB insertion.

There could be several reasons why it's failing:

1. Cosmos DB is rejecting the incoming connection - for example due to firewall rules or invalid connection strings.
2. There is a problem with the document data being inserted.

The error log also doesn't provide a detailed error message which makes it harder to fault-find. To add more detailed logging messages, you can use the following improvement for the PostAPI function:

```

1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;

```

```

10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12
13 namespace PostAPI
14 {
15     public static class PostAPI
16     {
17         private static readonly HttpClient httpClient = new
18             ↪ HttpClient();
19         private static readonly string cosmosDbEndpoint =
20             ↪ Environment.GetEnvironmentVariable("DBENDPOINT",
21             ↪ EnvironmentVariableTarget.Process);
22         private static readonly string cosmosDbKey = Environment.
23             ↪ GetEnvironmentVariable("DBKEY",
24             ↪ EnvironmentVariableTarget.Process);
25         private static readonly string databaseId = Environment.
26             ↪ GetEnvironmentVariable("DBID",
27             ↪ EnvironmentVariableTarget.Process);
28         private static readonly string containerId = Environment.
29             ↪ GetEnvironmentVariable("CONTAINERID",
30             ↪ EnvironmentVariableTarget.Process);
31         private static readonly CosmosClient cosmosClient = new
32             ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey, new
33             ↪ CosmosClientOptions() { ApplicationName = "FiDelApp"
34             ↪ });
35         private static readonly Container cosmosContainer =
36             ↪ cosmosClient.GetContainer(databaseId, containerId);
37
38         [FunctionName("FiDelApp")]
39         public static async Task<IActionResult> Run(
40             [HttpTrigger(AuthorizationLevel.Anonymous, "post",
41             ↪ Route = null)] HttpRequest req,
42             ILogger log)
43         {
44             log.LogInformation("Received a PostAPI request.");
45
46             string requestBody = await new StreamReader(req.
47             ↪ Body).ReadToEndAsync();
48             log.LogInformation($"Request body: {requestBody}");
49
50             try
51             {
52                 var requestDocument = JsonConvert.
53             ↪ DeserializeObject<RequestDocument>(requestBody);
54                 if (string.IsNullOrEmpty(requestDocument.id))
55                 {
56                     log.LogWarning("The 'id' property is
57             ↪ required.");
58                 }
59             }
60         }
61     }
62 }

```

```

41         return new BadRequestObjectResult("The 'id'
↪ property is required.");
42     }
43     log.LogInformation($"Inserting data into Cosmos
↪ DB: {JsonConvert.SerializeObject(requestDocument)}");
44     var response = await cosmosContainer.
↪ CreateItemAsync(requestDocument);
45     log.LogInformation("Data inserted to Cosmos DB
↪ successfully.");
46     return new OkObjectResult("Data inserted
↪ successfully! Well Done! (12/07/2023)");
47     }
48     catch (CosmosException ex)
49     {
50         log.LogError(ex, "A Cosmos DB error occurred.")
↪ ;
51         throw;
52     }
53     catch (Exception ex)
54     {
55         log.LogError(ex, "An error occurred.");
56         return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
57     }
58 }
59 }
60
61 public class RequestDocument
62 {
63     public string id { get; set; }
64     public string categoryId { get; set; }
65     public string categoryName { get; set; }
66     public string sku { get; set; }
67     public string name { get; set; }
68     public string description { get; set; }
69     public double Price { get; set; }
70 }
71 }

```

Addressing the two potential causes:

1. We can use the CosmosClientOptions to set the ApplicationName while creating the CosmosClient which will be useful while tracing the logs on the Cosmos DB side.
2. I added detailed logging messages to trace all the steps in the function from receiving the request, to parsing and inserting it which

will provide more information in case of failures. In addition, I separated 'CosmosException' from 'Exception' to distinguish issues about Cosmos DB specifically.

F.3 Prompt Test using both user and system message

DevOptimizeAI response for GetMean

Based upon the JMeter test results and the provided source code, it appears the main problem is in getting the data from the Cosmos DB. Most calls result in a 404 status code, indicating the requested resource is not found on the server. This seems to be due to a failure to find the requested ID in the Cosmos DB document store.

The given function is an Azure Function that is triggered by an HTTP GET request, retrieves an item in a specified partition of the Cosmos DB based on a provided ID, and returns the result to the client. It ends up catching a CosmosException when the requested item is not found, which is when it returns the 404 NotFound status. The function also catches any other exceptions and sends back a 500 Internal Server Error status.

A good improvement to the source code will be to add logging to the successful operation of the REST call, to provide visibility into the execution of the function. Second, creating a cache that will save the data after first request to the Cosmos DB will prevent the expensive operation of calling Cosmos DB every single time when we need to get a record.

Thus, the following is the updated function code:

```
1 public static class GetWebAPI
2 {
3     private static readonly ConcurrentDictionary<string,
4         ↳ RequestDocument> cache = new ConcurrentDictionary<
5         ↳ string, RequestDocument>();
6     // ... Rest of the objects declaration
7     [FunctionName("GetWebAPI")]
8     public static async Task<IActionResult> Run(
9         [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route
10         ↳ = null)] HttpRequest req,
11         ILogger log)
```

```

12     {
13         string id = req.Query["id"];
14
15         // Validate that the ID is provided
16         if (string.IsNullOrEmpty(id))
17         {
18             return new BadRequestObjectResult("The 'id'
↪ query parameter is required.");
19         }
20
21         // Check if item in cache
22         if (cache.TryGetValue(id, out var cachedItem))
23         {
24             log.LogInformation($"Retrieved cached item with
↪ ID '{id}'.");
25             return new OkObjectResult(cachedItem);
26         }
27
28         // Retrieve the item from Cosmos DB
29         var response = await cosmosContainer.ReadItemAsync<
↪ RequestDocument>(id, new PartitionKey(id));
30
31         // Add retrieved item to cache
32         cache.TryAdd(id, response.Resource);
33
34         log.LogInformation($"Retrieved item with ID '{id}'
↪ from Cosmos DB.");
35         return new OkObjectResult(response.Resource);
36     }
37     catch (CosmosException ex) when (ex.StatusCode ==
↪ System.Net.HttpStatusCode.NotFound)
38     {
39         return new NotFoundResult();
40     }
41     catch (Exception ex)
42     {
43         log.LogError(ex, "An error occurred while
↪ retrieving data from Cosmos DB.");
44         return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
45     }
46 }
47 }

```

The cache policy in this example is simple and does not include expiration. In a real life scenario, more advanced caching policies should be used to prevent serving outdated data.

DevOptimizeAI response for GetAPI

Based upon the JMeter test results and the provided source code, it appears the main problem is in getting the data from the Cosmos DB. Most calls result in a 404 status code, indicating the requested resource is not found on the server. This seems to be due to a failure to find the requested ID in the Cosmos DB document store.

The given function is an Azure Function that is triggered by an HTTP GET request, retrieves an item in a specified partition of the Cosmos DB based on a provided ID, and returns the result to the client. It ends up catching a CosmosException when the requested item is not found, which is when it returns the 404 NotFound status. The function also catches any other exceptions and sends back a 500 Internal Server Error status.

A good improvement to the source code will be to add logging to the successful operation of the REST call, to provide visibility into the execution of the function. Second, creating a cache that will save the data after first request to the Cosmos DB will prevent the expensive operation of calling Cosmos DB every single time when we need to get a record.

Thus, the following is the updated function code:

```
1 public static class GetWebAPI
2 {
3     private static readonly ConcurrentDictionary<string,
4         ↳ RequestDocument> cache = new ConcurrentDictionary<
5         ↳ string, RequestDocument>();
6     // ... Rest of the objects declaration
7
8     [FunctionName("GetWebAPI")]
9     public static async Task<IActionResult> Run(
10        [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route
11        ↳ = null)] HttpRequest req,
12        ILogger log)
13    {
14        try
15        {
16            string id = req.Query["id"];
17
18            // Validate that the ID is provided
19            if (string.IsNullOrEmpty(id))
20            {
21                return new BadRequestObjectResult("The 'id'
22                ↳ query parameter is required.");
23            }
24        }
25    }
26 }
```



```

20
21     // Check if item in cache
22     if (cache.TryGetValue(id, out var cachedItem))
23     {
24         log.LogInformation($"Retrieved cached item with
↪ ID '{id}'");
25         return new OkObjectResult(cachedItem);
26     }
27
28     // Retrieve the item from Cosmos DB
29     var response = await cosmosContainer.ReadItemAsync<
↪ RequestDocument>(id, new PartitionKey(id));
30
31     // Add retrieved item to cache
32     cache.TryAdd(id, response.Resource);
33
34     log.LogInformation($"Retrieved item with ID '{id}'
↪ from Cosmos DB.");
35     return new OkObjectResult(response.Resource);
36 }
37 catch (CosmosException ex) when (ex.StatusCode ==
↪ System.Net.HttpStatusCode.NotFound)
38 {
39     return new NotFoundResult();
40 }
41 catch (Exception ex)
42 {
43     log.LogError(ex, "An error occurred while
↪ retrieving data from Cosmos DB.");
44     return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
45 }
46 }
47 }

```

The cache policy in this example is simple and does not include expiration. In a real life scenario, more advanced caching policies should be used to prevent serving outdated data.

DevOptimizeAI response for PostAPI

Based on the JMeter results and the provided source code, there are a couple of suggestions for enhancing and improving performance in the code:

1. Caching: CosmosDB retrieval maybe slow or unstable which results

in 500 errors. If the majority of the data you're retrieving from CosmosDB doesn't change frequently, you can cache your data. This can significantly improve your application's performance by reducing unnecessary I/O operations.

2. Error Handling: In your catch block, you're handling all types of exceptions in a very general manner. It would be better if you can try to anticipate the types of exceptions that might occur and handle each of them appropriately.
3. Database Connection: The database connection (CosmosDB in this case) creation process generally takes more time and efforts. Currently, a new database connection is created on every HTTP request, which is expensive in terms of resources. Making the database client as a singleton (one per application) might provide better results.
4. Load Testing: Based on the JMeter results, it seems that the application does not perform well under intense load. Load testing should be performed which help to identify performance bottlenecks and resolve those issues which improves the overall application performance.

Below is the improved code implementation:

```
1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10 using System.Net.Http;
11 using Microsoft.Azure.Cosmos;
12 using System.ComponentModel;
13 using Azure;
14 using System.Collections.Generic;
15
16 namespace PostAPI
17 {
18     public static class PostAPI
19     {
20         private static readonly HttpClient httpClient = new
21             ↪ HttpClient();
22
23         //Database Settings
```

```

23     private static readonly CosmosClient cosmosClient;
24     private static readonly Microsoft.Azure.Cosmos.
↪ Container cosmosContainer;
25
26     //Static constructor
27     static FiDelApp()
28     {
29         string cosmosDbEndpoint = Environment.
↪ GetEnvironmentVariable("DBENDPOINT",
↪ EnvironmentVariableTarget.Process);
30         string cosmosDbKey = Environment.
↪ GetEnvironmentVariable("DBKEY",
↪ EnvironmentVariableTarget.Process);
31         string databaseId = Environment.
↪ GetEnvironmentVariable("DBID",
↪ EnvironmentVariableTarget.Process);
32         string containerId = Environment.
↪ GetEnvironmentVariable("CONTAINERID",
↪ EnvironmentVariableTarget.Process);
33
34         cosmosClient = new CosmosClient(cosmosDbEndpoint,
↪ cosmosDbKey);
35         cosmosContainer = cosmosClient.GetContainer(
↪ databaseId, containerId);
36     }
37
38     [FunctionName("FiDelApp")]
39     public static async Task<IActionResult> Run(
40         [HttpTrigger(AuthorizationLevel.Anonymous, "post",
↪ Route = null)] HttpRequest req,
41         ILogger log)
42     {
43         try
44         {
45             string requestBody = await new StreamReader(req
↪ .Body).ReadToEndAsync();
46             var requestDocument = JsonConvert.
↪ DeserializeObject<RequestDocument>(requestBody);
47
48             if (string.IsNullOrEmpty(requestDocument.id))
49                 return new BadRequestObjectResult("The 'id'
↪ property is required.");
50
51             var response = await cosmosContainer.
↪ CreateItemAsync(requestDocument);
52             return new OkObjectResult("Data inserted
↪ successfully! Well Done! (12/07/2023)");
53         }

```

```

54     catch (CosmosException ex)
55     {
56         log.LogError(ex, $"CosmosDB service error.
↪ Error Details: { ex.Message }");
57         return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
58     }
59     catch(Exception ex)
60     {
61         log.LogError(ex, "An unexpected error occurred.
↪ ");
62         return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
63     }
64 }
65 }
66 }

```

This new implementation initiates the Cosmos client on the application start and reuses it for every request (Singleton). Also, it provides specialized error handling for `CosmosException`.

Appendix G

CPU-Intensive Test

G.1 CalculatePrimeNumbers Test

Original Function response

Based on your JMeter results, it seems like the performance of your API is pretty decent. The average response time is 47 milliseconds, and the maximum response time is 208 milliseconds. The standard deviation is also quite low. However, there is always room for improvement.

The main potential performance issue in your code lies in the GetPrimes() method. This method uses a simple but computationally expensive algorithm for generating prime numbers. To improve performance, you could use a more optimized algorithm like the Sieve of Eratosthenes:

Here's an optimized version of your code:

```
1 using System;
2 using System.Net;
3 using Microsoft.AspNetCore.Mvc;
4 using Microsoft.Extensions.Primitives;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using System.Collections.Generic;
9
10 public static class CalculatePrimeNumbers
11 {
12     [FunctionName("CalculatePrimeNumbers")]
13     public static IActionResult Run(
```

```

14     [HttpTrigger(AuthorizationLevel.Function, "get",
↳ Route = null)] HttpRequest req)
15     {
16         string number = req.Query["number"];
17
18         if (int.TryParse(number, out int num))
19         {
20             List<int> primes = GetPrimes(num);
21             return new OkObjectResult($"Prime numbers up to
↳ {num}: {string.Join(", ", primes)}");
22         }
23         else
24         {
25             return new BadRequestObjectResult("Please
↳ provide a valid integer.");
26         }
27     }
28
29     private static List<int> GetPrimes(int limit)
30     {
31         var primes = new List<int>();
32         var sieve = new bool[limit + 1];
33         for (int x = 2; x <= limit; x++)
34         {
35             if (!sieve[x])
36             {
37                 primes.Add(x);
38                 for (int u = 2*x; u <= limit; u += x)
39                 {
40                     sieve[u] = true;
41                 }
42             }
43         }
44         return primes;
45     }
46 }

```

In this optimized version of the code, the Sieve of Eratosthenes is used to find all primes less than or equal to the provided number. This algorithm works by iteratively marking the multiples of each prime number starting from 2. The multiples are not prime, so they are eliminated, leaving only the primes in the end. This algorithm is much faster, especially for larger ranges of numbers.

Error Injection test

The performance issues you're facing are likely due to the inefficient implementation of the prime number generation algorithm. This algorithm has a time complexity of $O(n^2)$, which makes it quite slow for larger numbers.

In addition, the HTTP 500 Internal Server Error indicates there is a problem with the server-side execution of the code. The error is likely caused by the division by zero error in this line: `if ((num % primes[i] == 0) || (num ↪ / (i-1) == 0))` When `i` is 1, `i-1` becomes 0 and causes a division by zero error.

Here's a revised version of the code that uses the Sieve of Eratosthenes algorithm, which is more efficient for generating prime numbers up to a given limit. Also, I've fixed the division by zero issue:

```
1 public static class CalculatePrimeNumbers
2 {
3     [FunctionName("CalculatePrimeNumbers")]
4     public static async Task<IActionResult> Run(
5         [HttpTrigger(AuthorizationLevel.Function, "get",
6 ↪ Route = null)] HttpRequest req, ILogger log)
7     {
8         log.LogInformation("C# HTTP trigger function
9 ↪ processed a request.");
10
11         string number = req.Query["number"];
12         string requestBody = await new StreamReader(req.
13 ↪ Body).ReadToEndAsync();
14         dynamic data = JsonConvert.DeserializeObject(
15 ↪ requestBody);
16         number = number ?? data?.number;
17
18         if (int.TryParse(number, out int num))
19         {
20             List<int> primes = GetPrimes(num);
21             return new OkObjectResult($"Prime numbers up to
22 ↪ {num}: {string.Join(", ", primes)}");
23         }
24         else
25         {
26             return new BadRequestObjectResult("Please
27 ↪ provide a valid integer.");
28         }
29     }
30
31     private static List<int> GetPrimes(int maxNum)
32     {
33         var primes = new List<int>();
34     }
35 }
```

```

28     var sieve = new bool[maxNum + 1];
29     for (int num = 2; num <= maxNum; num++)
30     {
31         if (!sieve[num])
32         {
33             primes.Add(num);
34             for (int i = num * 2; i <= maxNum; i += num
↪ )
35             {
36                 sieve[i] = true;
37             }
38         }
39     }
40     return primes;
41 }
42 }

```

This version of code should provide better performance and avoid server-side errors. However, keep in mind that generating a large number of prime numbers still requires significant computational resources. If the performance is still not satisfactory, you may need to consider a different approach or limit the maximum input value.

Infinite Loop test

The JTL results indicate that the HTTP requests made by the function are all failing with either a 502 (Bad Gateway) error or a 500 (Internal Server Error). The first thing to do is to handle these errors properly.

In addition, the function's performance could be improved by optimizing the GetPrimes function. The current implementation is not efficient because it checks every odd number to see if it's prime by checking divisibility by all smaller primes. A better approach would be to use the Sieve of Eratosthenes algorithm, which is more efficient for finding all primes up to a given limit.

Here is an improved version of your function:

```

1 using System;
2 using System.Net;
3 using Microsoft.AspNetCore.Mvc;
4 using Microsoft.Extensions.Primitives;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;

```



```

8 using System.Collections.Generic;
9 using System.IO;
10 using Newtonsoft.Json;
11
12 public static class CalculatePrimeNumbers
13 {
14     [FunctionName("CalculatePrimeNumbers")]
15     public static async Task<IActionResult> Run(
16         [HttpTrigger(AuthorizationLevel.Function, "get",
17         ↪ Route = null)] HttpRequest req, ILogger log)
18     {
19         log.LogInformation("C# HTTP trigger function
20         ↪ processed a request.");
21
22         string number = req.Query["number"];
23         string requestBody = await new StreamReader(req.
24         ↪ Body).ReadToEndAsync();
25         dynamic data = JsonConvert.DeserializeObject(
26         ↪ requestBody);
27         number = number ?? data?.number;
28
29         if (int.TryParse(number, out int num))
30         {
31             List<int> primes = GetPrimes(num);
32             return new OkObjectResult($"Prime numbers up to
33         ↪ {num}: {string.Join(", ", primes)}");
34         }
35         else
36         {
37             return new BadRequestObjectResult("Please
38         ↪ provide a valid integer.");
39         }
40     }
41
42     private static List<int> GetPrimes(int limit)
43     {
44         var primes = new List<int>();
45         var sieve = new bool[limit + 1];
46
47         for (int num = 2; num <= limit; num++)
48         {
49             if (!sieve[num])
50             {
51                 primes.Add(num);
52                 for (int multiple = num * 2; multiple <=
53         ↪ limit; multiple += num)
54                 {
55                     sieve[multiple] = true;
56                 }
57             }
58         }
59     }
60 }

```

```
49         }
50     }
51 }
52
53     return primes;
54 }
55 }
```

The function now uses the Sieve of Eratosthenes algorithm to efficiently calculate prime numbers. This should improve the performance of your function.

However, the 502 and 500 errors suggest there might be issues with the server or network that are not related to your function's code. You might need to investigate these issues separately.

Appendix H

Behavioural Test

H.1 Prompt Test providing prompt to different developer types

DevOptimizeAI response for Junior Developer

One of the ways to optimize your function is to create a query that will directly calculate the mean elevation for a specified country on the database side. This approach will reduce network overhead, as you will transfer a lot less data from the database to the function. Moreover, this should also reduce the CPU usage on the function side.

Here is the updated function:

```
1 using System;
2 using Microsoft.AspNetCore.Mvc;
3 using System.Threading.Tasks;
4 using Microsoft.Azure.WebJobs;
5 using Microsoft.Azure.WebJobs.Extensions.Http;
6 using Microsoft.AspNetCore.Http;
7 using Microsoft.Extensions.Logging;
8 using Microsoft.Azure.Cosmos;
9 using Microsoft.Azure.Cosmos.Linq;
10
11 namespace GetMean
12 {
13     public static class GetMean
14     {
15         private static readonly string cosmosDbEndpoint =
```

```

16 ↪ Environment.GetEnvironmentVariable("DBENDPOINT",
17 ↪ EnvironmentVariableTarget.Process);
    private static readonly string cosmosDbKey =
18 ↪ Environment.GetEnvironmentVariable("DBKEY",
19 ↪ EnvironmentVariableTarget.Process);
    private static readonly string databaseId =
20 ↪ Environment.GetEnvironmentVariable("DBID",
21 ↪ EnvironmentVariableTarget.Process);
    private static readonly string containerId =
22 ↪ Environment.GetEnvironmentVariable("CONTAINERID",
23 ↪ EnvironmentVariableTarget.Process);
    private static CosmosClient cosmosClient = new
24 ↪ CosmosClient(cosmosDbEndpoint, cosmosDbKey);
    private static Container cosmosContainer =
25 ↪ cosmosClient.GetContainer(databaseId, containerId);

26 [FunctionName("GetMean")]
27 public static async Task<IActionResult> Run(
28     [HttpTrigger(AuthorizationLevel.Anonymous, "get",
29     ↪ ", Route = null)] HttpRequest req,
30     ILogger log)
31     {
32         string country = req.Query["Country"];
33
34         if (string.IsNullOrEmpty(country))
35         {
36             ↪ return new BadRequestObjectResult("The '
37             ↪ Country' query parameter is required.");
38         }
39
40         IQueryable<double> elevationQuery =
41         ↪ cosmosContainer.GetItemLinqQueryable<Volcano>()
42
43         ↪ .Where(v => v.Country == country)
44
45         ↪ .Select(v => Convert.ToDouble(v.Elevation));
46
47         double sum = await elevationQuery.SumAsync();
48         long count = await elevationQuery.CountAsync();
49
50         if (count == 0)
51         {
52             ↪ return new NotFoundResult();
53         }
54
55         double meanElevation = Math.Round(sum / count,
56         ↪ 3);
57         ↪ return new OkObjectResult(meanElevation);

```

```

48     }
49 }
50
51 public class Volcano
52 {
53     public string VolcanoName { get; set; }
54     public string Country { get; set; }
55     public string Elevation { get; set; }
56 }
57 }

```

Please note that this can be done only if the Elevation property is stored as a number in the database. If it is stored as a string and a lot of your records have the elevation not set or set to a invalid number, this can lead to a performance degradation. In this case, you should have a separate process to clean and convert the data.

DevOptimizeAI response for Senior Developer

The JMeter results indicate a slow response time for the operation being performed. Your code is currently querying the database and iterating through each item, checking the country, before doing a calculation. This can potentially result in a large number of unnecessary operations, thus slowing down the process.

Another potential performance issue is with int parsing of the 'Elevation'. Although this is not significant, it does add up when iterating through a large dataset.

Let's optimize it with these steps:

1. Restructure the database query to only return items for the selected country. This can significantly reduce the number of items being processed.
2. Try to parse the 'Elevation' to an integer once and if it fails, store it as '0'. Log the warning message for debugging purposes but continue the execution.

Here is the adjusted code:

```

1 public static async Task<IActionResult> Run(
2     [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route
    ↪ = null)] HttpRequest req,

```

```

3     ILogger log)
4 {
5     try
6     {
7         string country = req.Query["Country"];
8         if (string.IsNullOrEmpty(country))
9         {
10            return new BadRequestObjectResult($"\"The '
↪ Country' query parameter is required.\");
11        }
12
13        FeedIterator<Volcano> queryResultSetIterator =
↪ cosmosContainer.GetItemQueryIterator<Volcano>(
14            new QueryDefinition($"SELECT * FROM c WHERE c.
↪ country = '{country}'")
15        );
16
17        double sum = 0;
18        int count = 0;
19
20        while (queryResultSetIterator.HasMoreResults)
21        {
22            FeedResponse<Volcano> currentResultSet = await
↪ queryResultSetIterator.ReadNextAsync();
23            foreach (Volcano volcano in currentResultSet)
24            {
25                int parsedElevation = 0;
26                if (int.TryParse(volcano.Elevation, out
↪ parsedElevation))
27                {
28                    sum += parsedElevation;
29                }
30                else
31                {
32                    log.LogWarning($"Could not parse
↪ elevation '{volcano.Elevation}' for volcano '{volcano.
↪ VolcanoName}'. This elevation will be marked as 0.");
33                }
34                count++;
35            }
36        }
37
38        if (count == 0)
39        {
40            return new NotFoundResult();
41        }
42
43        double meanElevation = Math.Round(sum / count, 3);

```

```

44     return new OkObjectResult(meanElevation);
45 }
46 catch (Exception ex)
47 {
48     log.LogError(ex, "An error occurred while
↪ retrieving data from Cosmos DB.");
49     return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
50 }
51 }

```

Remember to sanitize your inputs while creating the SQL query; else, it can lead to an SQL Injection attack. You may also want to handle exceptions or incorrect input at the SQL level.

Also, moving the cosmos client AllThread and connecting to a static field might cause stale connections and thread synchronization issues; if this is being reused across multiple threads. You can opt to make a new cosmos client for each request, or add connection pooling.

H.2 Prompt Test providing hint for the model

DevOptimizeAI response for GetMean

Analyzing your JTL results, I can see that your JMeter test made 100 requests, and your system was able to serve an average of 2.6 requests per second with an average server-side response time of 3589 milliseconds (ms) which seems a bit high and could be improved.

Looking at your function code, the main sections that may be contributing to the higher than desired response times could be the CosmosDB queries. On average, your application spends the majority of its time waiting for CosmosDB's response.

Therefore, one way to enhance the performance might be to adjust your CosmosDB querying strategy.

I would recommend to adapt the first version which was commented out and, thus, make the DB process the query instead of filtering in the application:

```

1 try
2 {
3     string country = req.Query["Country"];

```

```

4
5 if (string.IsNullOrEmpty(country))
6 {
7     return new BadRequestObjectResult("The 'Country' query
8     ↪ parameter is required.");
9 }
10 var sqlQueryText = $"SELECT c.Elevation FROM c WHERE c.
11     ↪ Country = '{country}'";
12 QueryDefinition queryDefinition = new QueryDefinition(
13     ↪ sqlQueryText);
14 FeedIterator<Volcano> queryResultSetIterator =
15     ↪ cosmosContainer.GetItemQueryIterator<Volcano>(
16     ↪ queryDefinition);
17
18 double sum = 0;
19 int count = 0;
20
21 while (queryResultSetIterator.HasMoreResults)
22 {
23     FeedResponse<Volcano> currentResultSet = await
24     ↪ queryResultSetIterator.ReadNextAsync();
25     foreach (Volcano volcano in currentResultSet)
26     {
27         if (int.TryParse(volcano.Elevation, out int
28         ↪ parsedElevation))
29         {
30             sum += parsedElevation;
31             count++;
32         }
33         else
34         {
35             log.LogWarning($"Could not parse elevation '{
36             ↪ volcano.Elevation}' for volcano '{volcano.VolcanoName
37             ↪ }'. This elevation will be ignored.");
38         }
39     }
40 }
41
42 if (count == 0)
43 {
44     return new NotFoundResult();
45 }
46
47 double meanElevation = Math.Round(sum / count, 3);
48 return new OkObjectResult(meanElevation);
49 }
50

```



```
43 catch (Exception ex)
44 {
45     log.LogError(ex, "An error occurred while retrieving data
        ↳ from Cosmos DB.");
46     return new StatusCodeResult(StatusCodes.
        ↳ Status500InternalServerError);
47 }
```

This highlighted function offloads the filtration of data to the CosmosDB, thus reducing the data transfer and parse computation.

Next to this, you should consider optimizing the way your application connects and communicates with the CosmosDB and ideally connection caching should be in place if possible. Consider indexing, partitioning, analyzing the collection RU/s and load. For example, make sure DB indexing is properly aligned with the queries could help.

Moreover, also take into consideration to handle peak load or traffic spikes, as I see your test started 10 threads but it finished only 3 of them. So make sure that your infrastructure and services are capable to handle the volume.

It is essential to have monitoring and observability to identify the bottlenecks and apply the proper solution.

H.3 Model behaviour with previous successful messages example

DevOptimizeAI response for GetMean

Analyzing your JTL results, I can see that your JMeter test made 100 requests, and your system was able to serve an average of 2.6 requests per second with an average server-side response time of 3589 milliseconds (ms) which seems a bit high and could be improved.

Looking at your function code, the main sections that may be contributing to the higher than desired response times could be the CosmosDB queries. On average, your application spends the majority of its time waiting for CosmosDB's response.

Therefore, one way to enhance the performance might be to adjust your CosmosDB querying strategy.

I would recommend to adapt the first version which was commented out and, thus, make the DB process the query instead of filtering in the application:

```
1 try
2 {
3     string country = req.Query["Country"];
4
5     if (string.IsNullOrEmpty(country))
6     {
7         return new BadRequestObjectResult("The 'Country' query
8         ↪ parameter is required.");
9     }
10
11     var sqlQueryText = $"SELECT c.Elevation FROM c WHERE c.
12     ↪ Country = '{country}'";
13     QueryDefinition queryDefinition = new QueryDefinition(
14     ↪ sqlQueryText);
15     FeedIterator<Volcano> queryResultSetIterator =
16     ↪ cosmosContainer.GetItemQueryIterator<Volcano>(
17     ↪ queryDefinition);
18
19     double sum = 0;
20     int count = 0;
21
22     while (queryResultSetIterator.HasMoreResults)
23     {
24         FeedResponse<Volcano> currentResultSet = await
25         ↪ queryResultSetIterator.ReadNextAsync();
26         foreach (Volcano volcano in currentResultSet)
27         {
28             if (int.TryParse(volcano.Elevation, out int
29             ↪ parsedElevation))
30             {
31                 sum += parsedElevation;
32                 count++;
33             }
34             else
35             {
36                 log.LogWarning($"Could not parse elevation '{
37                 ↪ volcano.Elevation}' for volcano '{volcano.VolcanoName
38                 ↪ }'. This elevation will be ignored.");
39             }
40         }
41     }
42
43     if (count == 0)
44     {
45         return new NotFoundResult();
46     }
47 }
```

```

37 }
38
39 double meanElevation = Math.Round(sum / count, 3);
40 return new OkObjectResult(meanElevation);
41 }
42
43 catch (Exception ex)
44 {
45     log.LogError(ex, "An error occurred while retrieving data
        ↪ from Cosmos DB.");
46     return new StatusCodeResult(StatusCodes.
        ↪ Status500InternalServerError);
47 }

```

This highlighted function offloads the filtration of data to the CosmosDB, thus reducing the data transfer and parse computation.

Next to this, you should consider optimizing the way your application connects and communicates with the CosmosDB and ideally connection caching should be in place if possible. Consider indexing, partitioning, analyzing the collection RU/s and load. For example, make sure DB indexing is properly aligned with the queries could help.

Moreover, also take into consideration to handle peak load or traffic spikes, as I see your test started 10 threads but it finished only 3 of them. So make sure that your infrastructure and services are capable to handle the volume.

It is essential to have monitoring and observability to identify the bottlenecks and apply the proper solution.

H.4 Prompt Test using feedback provided from the model for GetMean API

DevOptimizeAI response first feedback iteration

Your code seems to be read and written to CosmosDB synchronously which may result in blocking IO operation. Using asynchronous operations can help improve the performance in your scenario. Ergo, we can modify your code to utilize asynchronous programming to further enhance code performance.

Here's the improved implementation:

```
1 namespace GetMean
2 {
3 public static class GetMean
4 {
5 ...
6 [FunctionName("GetMean")]
7 public static async Task<IActionResult> Run(
8 ...
9 {
10 try
11 {
12 ...
13 double sum = 0;
14 int count = 0;
15
16         List<Task> tasks = new List<Task>();
17         while (queryResultSetIterator.HasMoreResults)
18         {
19             var resultTask = Task.Run(async () =>
20             {
21                 FeedResponse<Volcano> currentResultSet
22                 ↪ = await queryResultSetIterator.ReadNextAsync();
23                 foreach (Volcano volcano in
24                 ↪ currentResultSet)
25                 {
26                     if (int.TryParse(volcano.Elevation,
27                 ↪ out int parsedElevation))
28                     {
29                         lock (tasks)
30                         {
31                             sum += parsedElevation;
32                             count++;
33                         }
34                     }
35                     else
36                     {
37                         log.LogWarning($"Could not
38                 ↪ parse elevation '{volcano.Elevation}' for volcano '{
39                 ↪ volcano.VolcanoName}'. This elevation will be ignored.
40                 ↪ ");
41                     }
42                 }
43             });
44             tasks.Add(resultTask);
45         }
46     }
47 }
48 }
49 }
50 }
```

```

41         await Task.WhenAll(tasks);
42
43         ...
44     }
45     ...
46
47 }
48 }
49
50 public class Volcano
51 {
52     ...
53 }
54 }

```

In the updated code, we have:

1. Created a list of tasks.
2. On each iteration of while (`queryResultSetIterator.HasMoreResults`), we initiate a new task to process the data.
3. Used the `Task.Run` method to execute these operations in parallel.
4. Utilized `Task.WhenAll` to ensure the completion of all running tasks.
5. Applied a lock on the sum and count variables to handle synchronization.

The amount of improvement will depend on the number of items that are being processed. As a note, the lock can be a bottleneck if there are a lot of tasks trying to update the sum and count at the same time. If the count and sum are too large causing performance issues you may need to implement an other method like partition or map reduce.

DevOptimizeAI response second feedback iteration

From the JMeter test results, it is clear that the HTTP requests are failing with a 502 Bad Gateway error. This suggests that the backend server, possibly the Azure Function or even the Cosmos DB, is unable to process the request. This may be due to hitting throughput limits or other computational resources running out such as memory or CPU.

Here, the HTTP Requests are processed sequentially, the function makes an HTTP request for each country, which isn't very efficient. By implementing

parallel processing of those HTTP requests, we can make the function scale better under load and improve the overall latency of the function. Also, unwrapped exceptions in database operations can be handled better.

Here's an optimized version of your function where HTTP requests are processed concurrently:

```
1 public static class GetMean
2 {
3     ...
4
5     [FunctionName("GetMean")]
6     public static async Task<IActionResult> Run(
7         [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route
8             ↪ = null)] HttpRequest req,
9         ILogger log)
10    {
11        string country = req.Query["Country"];
12        if (string.IsNullOrEmpty(country))
13        {
14            return new BadRequestObjectResult("The 'Country'
15            ↪ query parameter is required.");
16        }
17
18        try
19        {
20            QueryDefinition queryDefinition = new
21            ↪ QueryDefinition("SELECT * FROM c WHERE c.Country =
22            ↪ @country")
23                .WithParameter("@country", country);
24
25            var tasks = new List<Task>();
26            var sum = 0.0;
27            var count = 0;
28            var queryResultSetIterator = cosmosContainer.
29            ↪ GetItemQueryIterator<Volcano>(queryDefinition);
30
31            while (queryResultSetIterator.HasMoreResults)
32            {
33                // Start a new task to process this set of
34                ↪ results
35                tasks.Add(Task.Run(async () =>
36                {
37                    var currentResultSet = await
38                    ↪ queryResultSetIterator.ReadNextAsync();
39
40                    foreach (var v in currentResultSet)
41                    {
```

```

35         if (int.TryParse(v.Elevation, out var
↪ parsedElevation))
36         {
37             Interlocked.Add(ref sum,
↪ parsedElevation);
38             Interlocked.Increment(ref count);
39         }
40         else
41         {
42             log.LogWarning($"Could not parse
↪ elevation '{v.Elevation}' for volcano '{v.VolcanoName
↪ }'. This elevation will be ignored.");
43         }
44     }
45     }));
46 }
47
48 // Wait for all tasks to complete
49 await Task.WhenAll(tasks);
50
51 if (count == 0)
52 {
53     return new NotFoundResult();
54 }
55
56 double meanElevation = Math.Round(sum / count, 3);
57
58 return new OkObjectResult(meanElevation);
59 }
60 catch (CosmosException cosmosEx) when (cosmosEx.
↪ StatusCode == HttpStatusCode.TooManyRequests)
61 {
62     log.LogError($"Request rate too large. Delay for {
↪ cosmosEx.RetryAfter}");
63     // Handle request rate too large
64     return new StatusCodeResult(StatusCodes.
↪ Status429TooManyRequests);
65 }
66 catch (Exception ex)
67 {
68     log.LogError(ex, "An error occurred while
↪ retrieving data from Cosmos DB.");
69     return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
70 }
71 }
72 }

```

Remember that with concurrency every async operation should be thread-safe. Here Interlocked methods are used to protect against concurrent updates of the sum and count variables.

JMeter test results show also Error 503 service unavailable which might be caused by other services your code is dependent on or insufficient resources of Azure Functions server.

Lastly, ensure to perform Load and Performance test to validate that this improved implementation manages application scaling requirements as required.

DevOptimizeAI response third feedback iteration

The error "502 Bad Gateway" indicates that the issue is most likely on the server side and not on the client's side. Based on the JTL results, the problem is persisting because of the HTTP requests getting "502 Bad Gateway" and "503 Service Unavailable" in response.

Coming to your Function Code, there are a few strategies you can apply to enhance its performance.

1. Reduce Latency: The more requests sent to the database for data retrieval, the more latency there will be. In your Function Code, you are getting data multiple times from the same database for all the volcanoes of a specific country. That could be improved by fetching it once and storing it locally.
2. Utilize Batch Processing: Rather than handling one request at a time, batch processing takes in multiple requests and processes them together.
3. Use Efficient Data Types: Efficient use of data types especially in case of huge data can help in consuming less memory and can increase the overall performance.
4. Retry Policy: It is a good practice to implement a retry policy in your application. Azure Cosmos DB SDK include a built-in retry policy which handles throttling by doing exponential back off.

Here is an example of how the function can be improved. Please note that you have to further modify this based on your specific requirements.

```
1 [FunctionName("GetMean")]
```



```

2 public static async Task<IActionResult> Run(
3 [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route =
4     ↪ null)] HttpRequest req,
5 ILogger log)
6 {
7     string country = req.Query["Country"];
8     if (string.IsNullOrEmpty(country))
9     {
10        return new BadRequestObjectResult("The 'Country' query
11            ↪ parameter is required.");
12    }
13
14    try
15    {
16        var queryDefinition = new QueryDefinition("SELECT *
17            ↪ FROM c WHERE c.Country = @country")
18            .WithParameter("@country", country);
19
20        // RequestOptions with a retry policy
21        var requestOptions = new RequestOptions { RetryPolicy =
22            ↪ new RetryPolicy(HttpStatusCode.TooManyRequests) };
23
24        FeedIterator<Volcano> feedIterator = cosmosContainer.
25            ↪ GetItemQueryIterator<Volcano>(queryDefinition,
26            ↪ requestOptions: requestOptions);
27
28        double sum = 0;
29        int count = 0;
30
31        while (feedIterator.HasMoreResults)
32        {
33            FeedResponse<Volcano> response = await feedIterator
34                ↪ .ReadNextAsync();
35            foreach (var v in response)
36            {
37                if (double.TryParse(v.Elevation, out var
38                    ↪ parsedElevation))
39                {
40                    sum += parsedElevation;
41                    count++;
42                }
43                else
44                {
45                    log.LogWarning($"Could not parse elevation
46                        ↪ '{v.Elevation}' for volcano '{v.VolcanoName}'. This
47                        ↪ elevation will be ignored.");
48                }
49            }
50        }
51    }
52 }

```

```

40     }
41
42     if (count == 0)
43     {
44         return new NotFoundResult();
45     }
46
47     double meanElevation = Math.Round(sum / count, 3);
48
49     return new OkObjectResult(meanElevation);
50 }
51 catch (Exception ex)
52 {
53     log.LogError(ex, "An error occurred while retrieving
↪ data from Cosmos DB.");
54     return new StatusCodeResult(StatusCodes.
↪ Status500InternalServerError);
55 }
56 }

```

Please note that the `RetryPolicy` class is deprecated and will be removed in future versions of the Azure Cosmos DB .NET SDK. It's recommended to use the new built-in retry logic by setting `CosmosClientOptions` when creating an instance of `CosmosClient`. We are still using it here for simplicity and it's better to refactor the retry mechanism in your production code.

Appendix I

ChatGPT Response

ChatGPT first response

User: provide an azure http trigger function in c# to get data from a CosmosDB NoSQL searching by country and compute the mean of the elevation

Model: Here is an example of how you can write an Azure Function with an HTTP trigger that fetches data from Azure Cosmos DB and calculates the mean of the elevation based on a provided country. Please replace [DatabaseName], [CollectionName], [ConnectionStringSetting] with your actual Cosmos DB details.

```
1 using System;
2 using System.Linq;
3 using System.Net;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Mvc;
6 using Microsoft.Azure.WebJobs;
7 using Microsoft.Azure.WebJobs.Extensions.Http;
8 using Microsoft.AspNetCore.Http;
9 using Microsoft.Extensions.Logging;
10 using Newtonsoft.Json;
11 using Microsoft.Azure.Documents.Client;
12 using Microsoft.Azure.Documents.Linq;
13
14 public static class GetMeanElevation
15 {
16     [FunctionName("GetMeanElevation")]
17     public static async Task<IActionResult> Run(
18         [HttpTrigger(AuthorizationLevel.Function, "get", "
```

```

19     ↪ post", Route = null)] HttpRequest req,
20         ILogger log)
21     {
22         log.LogInformation("C# HTTP trigger function
23         ↪ processed a request.");
24
25         string country = req.Query["country"];
26
27         string requestBody = await new StreamReader(req.
28         ↪ Body).ReadToEndAsync();
29         dynamic data = JsonConvert.DeserializeObject(
30         ↪ requestBody);
31         country = country ?? data?.country;
32
33         if (string.IsNullOrEmpty(country))
34         {
35             ↪ return new BadRequestObjectResult("Please pass
36             ↪ a country on the query string or in the request body")
37             ↪ ;
38         }
39
40         var client = new DocumentClient(new Uri("https://[
41         ↪ YourCosmosDBAccountName].documents.azure.com:443/"), "[
42         ↪ YourCosmosDBAccountKey]");
43         var query = client.CreateDocumentQuery<Document>(
44         ↪ UriFactory.CreateDocumentCollectionUri("[
45         ↪ DatabaseName]", "[CollectionName]"),
46         ↪ new FeedOptions { EnableCrossPartitionQuery =
47         ↪ true })
48         .Where(d => d.Country == country)
49         .AsDocumentQuery();
50
51         var elevations = new List<int>();
52         while (query.HasMoreResults)
53         {
54             ↪ foreach (var result in await query.
55             ↪ ExecuteNextAsync())
56             {
57                 ↪ if (result.Elevation != null)
58                 ↪ {
59                     ↪ elevations.Add(result.Elevation);
60                 }
61             }
62
63         if (elevations.Count == 0)
64         {
65             ↪ return new NotFoundResult();

```

```

56     }
57
58     var meanElevation = elevations.Average();
59
60     return new OkObjectResult($"The average elevation
↪ for {country} is {meanElevation}");
61 }
62 }

```

Listing I.1: first GPT-4 response

Please note that this code assumes that your Cosmos DB documents have a Country and Elevation properties. If your document structure is different, please adjust the query and property accesses accordingly. Also, don't forget to add the necessary NuGet packages for Microsoft.Azure.WebJobs.Extensions.CosmosDB and Microsoft.Azure.DocumentDB.Core.

User: *is the function efficient and correct?*

Model: *The function is correct in terms of achieving the required functionality of getting data from CosmosDB and calculating the mean of the elevation. It is efficient in the sense that it's using the IDocumentQuery interface which allows for asynchronous iteration over the results and handles pagination behind the scenes.*

However, the function could be improved for efficiency in a few ways:

- 1. Using Dependency Injection: The DocumentClient instance could be injected into the function using Dependency Injection, rather than creating a new instance for each function execution. This would allow for better connection management and performance.*
- 2. Handling Large Data: If the data from CosmosDB is large, fetching all data to the function and then calculating the average might not be efficient. It would be better to calculate the sum and count separately in the CosmosDB query itself, and then calculate the average in the function.*
- 3. Avoiding Null Checks: The null check for Elevation in the loop can be avoided if the data in CosmosDB is guaranteed to have an Elevation for every document.*
- 4. Input Validation: The function currently accepts both GET and POST requests. It would be more efficient to accept only GET requests with the country as a query parameter, unless there's a requirement to accept the country in the request body.*

5. *Partition Key: The function is currently using cross partition queries which can be expensive. If the Country is the partition key, we can execute queries within a partition which will be faster and cost-efficient.*

Remember to replace placeholders in the URI and Key with your actual Cosmos DB instance details. Also, the query is assuming that your documents have a property named "Country". If your document structure is different, please adjust accordingly.

ChatGPT enhanced response

User: *provide a better implementation with better performance*

Model: *Here's an improved version of the function that uses Dependency Injection to manage the DocumentClient instance. It also changes the HTTP trigger to only accept GET requests and moves the country to a route parameter. The CosmosDB query is also optimized to only select the Elevation field, reducing the amount of data transferred.*

You need to add a Startup.cs file to your project to set up Dependency Injection:

```
1 using Microsoft.Azure.Functions.Extensions.  
  ↪ DependencyInjection;  
2 using Microsoft.Extensions.DependencyInjection;  
3 using Microsoft.Azure.Documents.Client;  
4  
5 [assembly: FunctionsStartup(typeof(MyNamespace.Startup))]  
6  
7 namespace MyNamespace  
8 {  
9     public class Startup : FunctionsStartup  
10    {  
11        public override void Configure(  
12        ↪ IFunctionsHostBuilder builder)  
13        {  
14            builder.Services.AddSingleton((s) => {  
15                ↪ return new DocumentClient(new Uri("https  
16                ↪ ://[YourCosmosDBAccountName].documents.azure.com:443/"  
17                ↪ ), "[YourCosmosDBAccountKey]");  
18            });  
19        }  
20    }  
21 }
```

And here's the improved function:

```
1 using System;
2 using System.Linq;
3 using System.Net;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Mvc;
6 using Microsoft.Azure.WebJobs;
7 using Microsoft.Azure.WebJobs.Extensions.Http;
8 using Microsoft.AspNetCore.Http;
9 using Microsoft.Extensions.Logging;
10 using Newtonsoft.Json;
11 using Microsoft.Azure.Documents.Client;
12 using Microsoft.Azure.Documents.Linq;
13
14 public static class GetMeanElevation
15 {
16     private static DocumentClient client;
17
18     public GetMeanElevation(DocumentClient documentClient)
19     {
20         client = documentClient;
21     }
22
23     [FunctionName("GetMeanElevation")]
24     public static async Task<IActionResult> Run(
25         [HttpTrigger(AuthorizationLevel.Function, "get",
26 ↪ Route = "meanelevation/{country}")] HttpRequest req,
27         string country,
28         ILogger log)
29     {
30         log.LogInformation("C# HTTP trigger function
31 ↪ processed a request.");
32
33         if (string.IsNullOrEmpty(country))
34         {
35             return new BadRequestObjectResult("Please pass
36 ↪ a country on the query string or in the request body")
37 ↪ ;
38         }
39
40         var query = client.CreateDocumentQuery<int>(
41             UriFactory.CreateDocumentCollectionUri("[
42 ↪ DatabaseName]", "[CollectionName]"),
43             new SqlQuerySpec
44             {
45                 QueryText = "SELECT VALUE c.Elevation FROM
46 ↪ c WHERE c.Country = @country",
```

```

41         Parameters = new SqlParameterCollection()
42         {
43             new SqlParameter("@country", country)
44         }
45     },
46     new FeedOptions { EnableCrossPartitionQuery =
↪ true })
47     .AsDocumentQuery();
48
49     var elevations = new List<int>();
50     while (query.HasMoreResults)
51     {
52         foreach (var result in await query.
↪ ExecuteNextAsync())
53         {
54             elevations.Add(result);
55         }
56     }
57
58     if (elevations.Count == 0)
59     {
60         return new NotFoundResult();
61     }
62
63     var meanElevation = elevations.Average();
64
65     return new OkObjectResult($"The average elevation
↪ for {country} is {meanElevation}");
66 }
67 }

```

Please note that this code assumes that your Cosmos DB documents have a Country and Elevation properties. If your document structure is different, please adjust the query and property accesses accordingly. Also, don't forget to add the necessary NuGet packages for Microsoft.Azure.WebJobs.Extensions.CosmosDB and Microsoft.Azure.DocumentDB.Core.

User: is the function efficient and correct?

Model: The given function is correct in terms of functionality as it fetches data from Cosmos DB based on a given country and calculates the mean of the elevation. It is efficient in the sense that it's using the IDocumentQuery interface which allows for asynchronous iteration over the results and handles pagination behind the scenes.

However, the function could be improved for efficiency in a few ways:

- *Using Dependency Injection: The DocumentClient instance is injected into the function using Dependency Injection, rather than creating a new instance for each function execution. This would allow for better connection management and performance.*
- *Handling Large Data: If the data from Cosmos DB is large, fetching all data to the function and then calculating the average might not be efficient. It would be better to calculate the sum and count separately in the Cosmos DB query itself, and then calculate the average in the function.*
- *Avoiding Null Checks: The null check for Elevation in the loop can be avoided if the data in Cosmos DB is guaranteed to have an Elevation for every document.*
- *Input Validation: The function currently accepts only GET requests with the country as a route parameter.*
- *Partition Key: The function is currently using cross partition queries which can be expensive. If the Country is the partition key, we can execute queries within a partition which will be faster and cost-efficient.*

Remember to replace placeholders in the URI and Key with your actual Cosmos DB instance details. Also, the query is assuming that your documents have a property named "Country". If your document structure is different, please adjust accordingly.