MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Leveraging Large Language Models to Automatically Infer RESTful API Specifications

DECROP, Alix

Award date:
2023

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

Leveraging Large Language Models to Automatically Infer RESTful API Specifications

DECROP Alix

..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : Prof. DEVROEY Xavier

Co-promoteur : Prof. PERROUIN Gilles

Mémoire présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à finalité spécialisée en Software Engineering

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR (BELGIUM)

Acknowledgments

I would like to acknowledge the professors, mentors, friends and family members that have helped me thoroughly during my Master's Thesis.

First, I would like to thank my promotor and co-promotor, Prof. Xavier Devroey and Prof. Gilles Perrouin respectively. They have guided me from the beginning of the Master's Thesis until the very end with various meetings and crucial insight.

Second, I would like to thank members of the *SnT* with whom I worked with during 4 months: Ahmed Khanfir, who regularly helped and assisted me with the overall development of this Master's Thesis. Mike Papadakis, who brought major insight regarding the initial refining of this Master's Thesis and thereafter key recommendations. Renzo Degiovanni, who welcomed me to the research center and with whom I regularly discussed the ongoing work.

Third, I would like to thank friends and family members who have encouraged me during my Master's Thesis. I would especially like to thank Valentina and Gauthier who have helped me regarding the cost of leveraging models, and the student interns at the *SnT* with whom I had various interesting conversations.

Abstract

Application Programming Interfaces, known as APIs, are increasingly popular in modern web applications. With APIs, users around the world are able to access a plethora of data contained in numerous server databases. To understand the workings of an API, a formal documentation is required. This documentation is also required by API testing tools, aimed at improving the reliability of APIs. However, as writing API documentations can be time-consuming, API developers often overlook the process, resulting in unavailable, incomplete or informal API documentations.

Recent Large Language Model technologies such as *ChatGPT* have displayed exceptionally efficient capabilities at automating tasks, disposing of data trained on billions of resources across the web. Thus, such capabilities could be utilized for the purpose of generating API documentations.

Therefore, the Master's Thesis proposes the first approach **Leveraging Large Language Models to Automatically Infer RESTful API Specifications**. Preliminary strategies are explored, leading to the implementation of a tool entitled *MutGPT*. The intent of *MutGPT* is to discover API features by generating and modifying valid API requests, with the help of Large Language Models.

Experimental results demonstrate that *MutGPT* is capable of sufficiently inferring the specification of the tested APIs, with an average route discovery rate of 82.49% and an average parameter discovery rate of 75.10%. Additionally, *MutGPT* was capable of discovering 2 undocumented and valid routes of a tested API, which has been confirmed by the relevant developers.

Overall, this Master's Thesis uncovers 2 new contributions:

1. Large Language Models are capable of generating valid and diversified HTTP requests for RESTful APIs, only requiring the name of the API as input.

2. It is possible to automatically infer RESTful API specifications by leveraging Large Language Models.

A replication package with the implementation and evaluation data is available at the following *GitHub* repository URL: `https://github.com/alixdecr/MutGPT`**.**

Keywords*: RESTful APIs, Large Language Models, HTTP Requests, Masking, Mutation*

Résumé

Les interfaces de programmation d'applications, connues sous le nom d'APIs, sont de plus en plus populaires dans les applications web modernes. Grâce aux APIs, les utilisateurs du monde entier peuvent accéder à une immense quantité de données contenues dans de nombreuses bases de données de serveurs. Pour comprendre le fonctionnement d'une API, une documentation formelle est requise. Cette documentation est également requise par les outils de test d'APIs, qui visent à améliorer leur fiabilité. Cependant, comme la rédaction de documentations portant sur les APIs peut prendre beaucoup de temps, les développeurs d'APIs négligent souvent ce processus, ce qui résulte en des documentations non-disponibles, incomplètes ou informelles.

Les technologies récentes de grands modèles de langage - Large Language Models en anglais - telles que *ChatGPT* ont démontré une efficacité exceptionnelle dans l'automatisation de divers tâches, disposant de données entrainées sur des milliards de ressources à travers le web. De ce fait, ces capacités pourraient être utilisées pour générer des documentations d'APIs.

Pour cette raison, ce mémoire de Master propose la première approche **utilisant les grands modèles de langage pour automatiquement inférer les spécifications d'APIs RESTful**. Des stratégies préliminaires sont explorées, menant à l'implémentation d'un outil intitulé *MutGPT*. L'objectif de *MutGPT* est de découvrir les caractéristiques d'APIs en générant et en modifiant des requêtes d'APIs valides, à l'aide de grands modèles de langage.

Les résultats expérimentaux démontrent que *MutGPT* est capable d'inférer suffisamment les spécifications des APIs testées, avec un taux moyen de découverte de routes de 82,49% et un taux moyen de découverte de paramètres de 75,10%. De plus, *MutGPT* fut capable de découvrir 2 routes non documentées et valides d'une API testée, ce qui a été confirmé par les développeurs concernés.

Dans l'ensemble, ce mémoire de Master présente 2 nouvelles contributions:

1. Les grands modèles de langage sont capables de générer des requêtes HTTP valides et diversifiées pour des APIs RESTful, ne nécessitant que le nom de l'API en entrée.

2. Il est possible d'inférer automatiquement les spécifications des APIs RESTful en utilisant des grands modèles de langage.

Un paquet de réplication contenant les données d'implémentation et d'évaluation est disponible sur un dépôt *GitHub* à l'adresse suivante : `https://github.com/alixdecr/MutGPT`**.**

***Mots-clés**: APIs RESTful, Grands modèles de langage, Requêtes HTTP, Masquer, Muter*

Table of Contents

1 Introduction

1.1 The Popularity and Importance of APIs

In today's interconnected era, software programs rarely operate in isolation. Instead, programs often rely on each other, sending and receiving various pieces of data in order to deliver distinct services to their users. This type of software architecture allows a multitude of programs to collect and manipulate data from a single host server, highly reducing the redundancy of data in storage. Accordingly, this service allowing the communication between software applications is called an API, short for Application Programming Interface. With the rise of new technologies, APIs are increasingly common, allowing developers around the world to access a single point containing a plethora of data.

Since APIs are extremely popular, their reliability is of foremost importance to assure an adequate service delivery. Indeed, if a popular API utilized by thousands of different programs is unreliable, it could lead to disastrous consequences. Such examples of this matter exist. In September 2018, a bug was discovered in the *Facebook* API regarding shared photos. The defect would potentially allow developers to access private pictures of users, despite solely giving permission to access photos shared on their timelines. These accidentally divulged private pictures include those shared on *Facebook Marketplace* and *Facebook Stories*, 2 features of the social media. Moreover, the bug also uncovered private pictures that users uploaded on *Facebook*, which were not officially posted. According to *Facebook*, this photo API bug affected approximately 6.8 million users and 1,500 software applications built by 876 developers [19]. To give an order of magnitude, *Facebook* had over 2,320 million monthly active users - worldwide that is - by the last quarter of 2018 [32]. Thus, the importance of maintaining a high reliability and security of popular APIs is a crucial matter.

For this purpose, various testing tools designed especially for APIs have been - and are being - developed. These testing tools can be effective at finding bugs in APIs, thus increasing the reliability of the tested APIs. A well-known example in the domain is *Postman* [65], a full-on and widely used platform to build and test APIs. Another testing tool, *RESTest* [71], allows developers to test APIs in a robot ecosystem. A last example is *RESTler* [72], an automatic fuzzing tool able to find security and reliability bugs in REST APIs.

1.2 The Value of API Documentations

Even though API testing tools can be effective, the process comes with a catch: A formal description of the API under test is required, in order to generate adequate test cases for it. This description is often documented in a format known as the *OpenAPI* specification [61], formerly known as the *Swagger* specification of an API. This popular format allows testers to implement testing programs that are able to comprehend routes, parameters and other important features of tested APIs.

Not only are API documentations useful for testing purposes, such documentations are also of foremost importance for developers leveraging APIs. Indeed, in order for a developer to utilize an API, the developers needs to know how the API functions, what routes it contains and what specific parameters are usable in API requests.

However, in a world where APIs are increasingly common [33], software developers often overlook the process of documenting APIs. The alleviation of this tedious task often results in poorly documented APIs, incomplete API documentations or even APIs that are not documented at all. Some API websites might contain a well-made and intelligible documentation for a human, but which is written in an informal manner and thus impossible to interpret by the relevant testing tools. To exemplify, if an individual purchases a brand new bed to construct that does not contain instructions, the building process will be tedious and difficult.

In consequence, an API documentation is indispensable for developers to understand the API they are leveraging and for API testing tools requiring a formal specification of the API in order to test it.

1.3 The Baseline of APIs: Requests and Responses

In order to retrieve data from an API, developers can send to API servers what is known as requests. A request contains a formal message, describing a resource to be retrieved or an action to be performed. Moreover, both the client and the server understand this formal message, as it follows a set of standardized rules known as HTTP - acronym for Hypertext Transfer Protocol.

When receiving requests, API servers will analyze their content. If the server is able to understand a request, it will respond to the client accordingly. However, if the server does not understand a given request, it will also respond to the client but with content notifying that the sent request was not understood. Server responses are understood by the client, as it also follows the rules of HTTP.

Thus, HTTP requests and responses consist in the baseline for manipulating APIs.

1.4 Leveraging Large Language Models to Discover APIs

With the rise of Large Language Models such as the notorious *ChatGPT*, automating straightforward tasks becomes an undemanding task in itself. Large Language Models - abbreviated as LLMs following this paragraph - are seeing a lot of modern use in various domains, especially in computer science. The performance of such models is a consequence of the enormous quantity of training data used, allowing models to predict and understand human-like text. For instance, a user can ask a Large Language Model to write a trivial piece of code, to which a valid answer and a description of the answer are correctly returned by the model in most cases.

As Large Language Models have astonishing capabilities at automating tasks with the help of vast training data, a question was asked: What if a model such as *ChatGPT* could be able to automatically discover API information, only requiring the user to input the name of the relevant API? The automation of such task would allow developers to perceive the usage of APIs, without having to rely on existing documentations. Valid API actions, routes, parameters and requests could be discovered, alleviating a demanding task in the API domain.

To conclude, a process as undemanding as snapping fingers is worthy of being explored. For this reason, **Leveraging Large Language Models to Automatically Infer RESTful API Specifications** is the subject of this Master's Thesis, which the following of this document will explore and explain.

1.5 Thesis Plan

The content of the document is organized in the following sections:

Section 2 presents the **State of the Art**. The section will first familiarize the reader with basic knowledge relevant to the work, then present existing work surrounding API testing and Large Language Models.

Section 3 details the **Motivation** of the Thesis. The section will present the initial research that was carried out, in order to discover the practical use of Large Language Models. Then, preliminary findings will present the different strategies explored for the purpose of generating API information with Large Language Models. The section will conclude by presenting a new strategy, consisting in mutating API requests.

Section 4 contains a complete explanation of ***MutGPT*: The Automatic Request Mutation Tool**, developed for the purpose of this Master's Thesis. The section will present everything there is to know about the tool, spanning from a general overview of the process to a specific demonstration with an existing API. An improvement of the tool is also exhibited in the section.

Section 5 demonstrates the **Evaluation and Results**, the section comprising 3 research questions aimed at testing the effectiveness of the implemented approach. Each research question will contain experiments, consisting of a setup, metrics and observed results. The section terminates with a description of the encountered internal and external threats to validity.

Section 6 contains the **Discussion** regarding the work. The section will consist in discussing the obtained results of Section 5, and presenting the obtained feedback for undocumented results found with a tested API.

Section 7 presents the **Future Work** surrounding the tool. The section will also display possible improvements and upgrades for the implemented tool.

Section 8 winds up the Master's Thesis with a **Conclusion**. The complete research is summarized, and the main contributions of the work are communicated.

The **Acronyms** used throughout the document and the **Bibliography** containing all leveraged references consist in the last sections of the document.

2 State of the Art

To introduce the important concepts utilized for the purpose of this Master's Thesis, the current section will begin by detailing the relevant **Background** involved in the work. Then, state-of-the-art research and tools are detailed; **API Testing** and **Leveraging Large Language Models** are the two main points of interest.

2.1 Background

As the section presents definitions relative to the background of the work, the quoting of the utilized references is of foremost importance for reliability purposes.

The Master's Thesis focusing on Large Language Model technologies such as *ChatGPT*, its use in generating valid definitions was explored. Thus, each definition was prompted to *ChatGPT* and analyzed. However, as explained in Section 3.1.3, such technology can in certain cases generate invalid information. For this reason, each generated definition was compared to trustworthy and valid sources. By comparing definitions generated by *ChatGPT* and definitions from existing sources, it becomes possible to formulate varied and valid definitions. Consequently, each definition contained in a green rectangle is a personal formulation based on official definition sources and based on definitions generated by *ChatGPT*. The definitions will contain references of the consulted resources.

2.1.1 Application Programming Interfaces (APIs)

Even though the notion of an API is broadly known by most computer scientists, it is rarely defined in precise terms. In order to fully grasp the concept behind the acronym, an in-depth explanation regarding APIs is presented.

Definition 2.1: API

An **Application Programming Interface (API)** is a set of rules specifying how computer programs should interact with each other. It provides a standard interface for communication between different programs, allowing developers to integrate different functionalities into their own programs without needing to know how these functionalities are implemented.

Definition based on: [37] [78]

APIs are used extensively in modern software development, enabling developers to build new applications by leveraging various functionalities from other existing application services. Thus, using APIs increases efficiency, reduces development time, and improves the user experience by providing a complete and hidden integration between different software applications.

Without any other context, an API is a general term that can encompass different types of architectures, protocols and tools. For the purpose of this Master's Thesis, the focus will be driven towards a specific type of API entitled a REST - or RESTful - API.

Before detailing such APIs, the REST architecture on which they rely upon is explained.

Definition 2.2: REST

Representational State Transfer (REST) is a software architectural style for the World Wide Web, characterized by a set of constraints describing how web services are supposed to behave. These constraints include a *client-server architecture*, *statelessness*, *cacheability*, a *uniform interface*, and a *layered system*.

Definition based on: [26] [67]

Since REST is an architectural style and not a standard, it is therefore not standardized. Nonetheless, the defined constraints of the REST architecture provide a standardized approach for designing and implementing

web services. Thus, the implementation process of services following the REST architecture principles is very flexible, which allows their design to be scalable and flexible.

The REST constraints can be described as follows:

Client-Server Architecture. The client and the server are separated from each other and can evolve independently. The client sends requests to the server, and the server returns responses to the client. This constraint allows a single API to be used by multiple clients.

Statelessness. The server does not store any client context between requests. Thus, each client request should be self-contained with all the necessary information to achieve and process the request. This principle increases the availability of APIs following it.

Cacheability. Responses from the server can be cached by the client to improve performance. For instance, a client can store copies of frequently accessed data, which reduces the redundancy of client-server interactions.

Uniform Interface. The server provides a standardized interface that is independent of the client. This allows the decoupling of the client from the implementation of the REST service. Two main standards are used when sending requests, in order to define the interface:

- **Uniform Resource Identifiers (URIs)** allow the identification of specific resources by specifying a unique sequence of characters.
- **HTTP Methods** allow the identification of the operations that can be performed on those resources, using predefined methods. Such examples of these methods include the *GET* method which indicates the retrieval of a specified resource, or the *POST* method which indicates the submission of data to a specified resource. Information regarding HTTP requests, encompassing HTTP methods, is described in Section 2.1.3.

Layered System. The architecture is organized in layers, and each layer has a specific responsibility. With this constraint, a certain layer should not impact another layer. Moreover, a client should not be able to distinguish if it is connected to the end server directly or to an intermediary.

Definition 2.3: REST API

A **REST API** - also referred to as a **RESTful API** - is an Application Programming Interface following the design principles of the Representational State Transfer architectural style, providing a standardized way of communication between applications over the internet. REST APIs use HTTP requests to perform various operations on data resources identified by URIs.

Definition based on: [21] [80]

As said previously, the REST architecture is not a standard. Thus, APIs following the said architecture attempt to conform the most to each and every REST constraint. However, many APIs do not conform perfectly to these constraints, which has caused the term RESTful to describe such APIs.

2.1.2 Practical Example of a REST API: The Weather Forecast

Figure 1 illustrates an example of a REST API for a weather forecast service[1]. On the left, there are multiple clients that can interact with the API server, which is itself located on the right. As both parties are separated from each other and can evolve independently, the **Client-Server Architecture** constraint of REST is satisfied. Moreover, the example displays a client which would like to get the current weather information for the city of London.

In order to do so, the client sends a HTTP request of the GET method to the `/weather` route of the API server with a parameter `location=london`, indicating the retrieval of a resource. The client knows how to

[1] As the example API does not exist, utilized elements in the illustration such as the route and the response data do not exist either. However, the *OpenWeatherMap* API [63] is an existing example of a weather forecast API exhibiting a similar usage to the given example.

formulate requests as there is a **Uniform Interface** from the HTTP protocol, consisting in another REST constraint. As the server does not store any client context, the request is self-contained with all the information needed to execute the request. Thus, the **Statelessness** constraint is also achieved. Then, the server receives and analyzes the client request. Based on the request, the server is able to determine that the client requested data about the weather in London. The server retrieves this data from a weather database, and sends it back to the client based on the HTTP response structure. In this case, a `200 OK` status codes indicates that the request of the client has succeeded. The client can now access data about the weather in London, contained in the response.

For the **Cacheability** REST constraint, one could suppose that as London is a famous city, a client would like to request this information multiple times during the day. Thus, the client could store a copy of this frequently accessed data, for a certain period of time as the weather changes over time.

Furthermore, the weather API could be deployed on a server, and the weather database could be contained in another server. Thus the **Layered System** constraint is also satisfied. The cache described earlier can also consist in a layer of the REST API.

As all REST constraints are satisfied, the example exhibits correctly a representation of a REST weather forecast API.

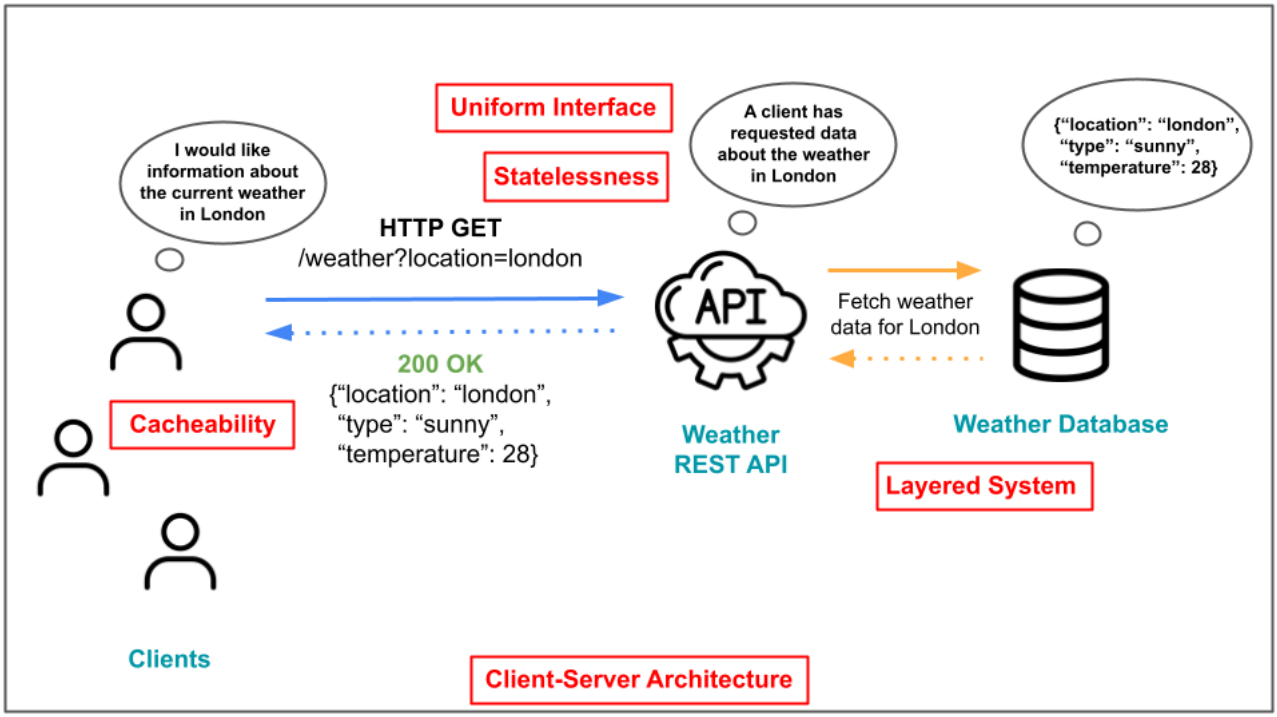


Figure 1: Example of a REST API for the weather forecast. REST constraints are represented in red.

2.1.3 HTTP Requests

As the core contribution of this Master's Thesis considerably focuses on REST API requests, the understanding of HTTP requests is of foremost importance. Prior to defining such requests, HTTP is characterized.

Definition 2.4: HTTP

The **Hypertext Transfer Protocol (HTTP)** is a standardized protocol used for communication between web clients and web servers. It defines a set of rules enabling the exchange of information on the Internet. HTTP follows a client-server model, where the client sends a request to the server, and the server responds with the requested information. HTTP is regarded as the foundation of data exchange on the World Wide Web. Moreover, the protocol facilitates tasks such as sending and receiving data, retrieving web pages and interacting with web-based applications in general.

Definition based on: [48]

As detailed in the definition, the Hypertext Transfer Protocol relies on a client-server architecture. For instance, a user leveraging a web browser such as *Chrome* or *Firefox* is a client in such model. The requested information or task specified by the client to the server is contained in what is known as a HTTP request.

Definition 2.5: HTTP Request

A **HTTP Request** is a message sent by a client to a server, specifying an action to be performed in the context of the Hypertext Transfer Protocol. It is a fundamental component of web communication and is used to retrieve or send data over the internet. A HTTP request embodies various components such as *request lines*, *header fields*, *query strings* and an optional *message body*.

Definition based on: [31] [49]

In order to further understand the embodiment of a HTTP request, each component is detailed in the following paragraphs.

Request Lines. The first element to consider in a request is denominated as a request line. Encompassed in a HTTP request, a request line also encompasses the following elements:

- A **HTTP Method** which indicates the type of action to be performed.
- A **Request URI** which identifies the path of the resource for the HTTP method action to be performed.
- The **HTTP Version Number** specifying the version of HTTP to be used.

Example: Request Line of a Request

An example of a request line contained in a HTTP request is:

```
GET /data/mycontent.html HTTP/1.1
```

With:

Header Fields. In an HTTP request, header fields are used to provide additional information about the request being made or to modify the behavior of the server handling the request. Each field consists of a key-value pair of the form `name:value`, and are all included in a specific header section of the HTTP request. There exists various HTTP header fields, some examples including:

- **User-Agent** identifies the client application or browser making the request.
- **Cookie** allows the inclusion of a previously stored cookies for the purpose of maintaining a session.

- **Content-Length** specifies the length of the request body, measured in bytes.

Example: Header Fields of a Request

An example of a header contained in a HTTP request consists in the following fields:

```
User-Agent:  Mozilla/4.0
Accept-Language:  fr, en
Content-Length:  35
```

This header specifies that the web browser version `Mozilla/4.0` is the user agent, that French and English are accepted languages and that the content length is 35 bytes.

Message Body. Optional element of a HTTP request used to specify additional data. The message body is used to send data associated with the request such as the content of a form or the content of a file to upload. As HTTP requests containing the POST method are used to send data to the server, such requests will usually be associated with a corresponding message body, containing the relevant data to be posted. However, as most HTTP requests of the GET method ask data from the server, no message body is needed.

Example: Message Body of a Request

An example of a message body contained in a HTTP request of the POST method is:

```
{"name": "John Doe", "email": "johndoe@example.com"}
```

This example can represent user data (name and email) being sent to the server for an account registration on a website.

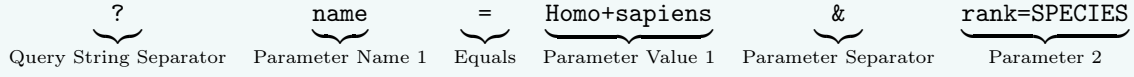
Query String. In order to specify various supplemental parameters, a request can contain a query string. This string represents a combination of parameter names and values, separated by ampersand `"&"` characters. A query string is added directly to the request URL, preceded by a question mark `"?"` character for the purpose of separating the parameters from the rest of the request.

Example: Query String of a Request

An example of a request containing a query string is given:

`https://api.gbif.org/v1/species?name=Homo+sapiens&rank=SPECIES`

By analyzing the content, the query string contains the following elements:



In the example above, the parameter value `Homo+sapiens` contains a `"+"` character. This is due to the fact that when a parameter value contains a spacing, it is encoded with either a `"+"` character or with `"%20"`. As a spacing cannot be contained in a URL, this step is mandatory in order to avoid an undesired splitting of a URL. Modern web browsers will automatically add such characters when a written URL contains spacings.

As GET HTTP requests sent to API endpoints usually need to specify a query string in order to retrieve specific data from the server, the manipulation of query strings is of foremost importance and will be utilized throughout the rest of the document.

Example: Complete HTTP Request

By assembling all defined request elements, a complete HTTP request would resemble the following structure:

```
POST /users/data HTTP/1.1

Host:  www.mywebsite.test
User-Agent:  Mozilla/4.0
Accept-Language:  fr, en
Content-Length:  45

{"name": "John Doe", "email": "johndoe@example.com"}
```

This example request indicates a POST method, signifying the server that data is sent. Various header fields regarding the request are specified. The sent message body is supposedly user data for a website registration.

Even though HTTP requests have been formally defined in the current section, their use will be limited to straightforwards and self-contained URLs for the research of this Master's Thesis. The only HTTP method used is the GET method, for the retrieval of relevant API data. As the research focuses on inferring API routes and parameters, generating GET HTTP request URLs is a sufficient task, as such URLs are able to contain all routes, parameter names and parameter values described in API specifications[2].

2.1.4 HTTP Responses

While the Hypertext Transfer Protocol encompasses requests that can be made from a client to a server, a server also replies to the client after receiving and analyzing a sent request.

Definition 2.6: HTTP Response

A **HTTP Response** is a message sent by a server in response to a HTTP request made by a client. It contains information about the status of the request and may include additional data or resources requested by the client, depending on the elements analyzed in the received request. A HTTP response typically contains 3 elements: a *status line*, *header fields* and a *message body*.

Definition based on: [24]

In order to further understand the embodiment of a HTTP response, each component is detailed in the following paragraphs.

Status Line. The first element of the response is called the status line. This line contains 3 items:

- The **HTTP Version Number** specifying the version of HTTP to be used.
- A **Status Code** representing the outcome of the request with a predefined number in the HTTP standard.
- A **Status Text** describing the meaning of the status code returned.

[2]Throughout the document, the terms API documentation, API specification and API grammar are interchangeable as they all relate to the description of an API.

Example: Status Line of a Response

An example of a status line contained in a HTTP server response is:

```
HTTP/1.1 201 Created
```

With:

In this example, a status code `201 Created` is returned in the response of the server, signifying that the sent request succeeded, resulting in the creation of a new resource.

Header Fields. Comparably to HTTP requests, HTTP responses can also contain header fields. Such fields provide additional information about the response being sent by the server, and communicate various details to the client regarding the nature of the response and how it should be handled accordingly.

Example: Header Fields of a Response

An example of a header contained in a HTTP response consists in the following fields:

```
Server:  Apache
Content-Type:  application/json
Content-Length:  128
```

This header specifies that the name of the server from which the response originates is `Apache`, that the response data is in a JSON format and that its content length is 128 bytes.

Message Body. In order to send data back to the client, it needs to be carried in a message body. The message body can contain various types of data such as HTML, JSON, or any other content type specified by the server in the header fields.

Example: Message Body of a Response

An example of a message body contained in a HTTP response answering to a GET HTTP request is:

```
{"name": "Whiskers", "age": 4, "breed": "Persian"}
```

This example represents data about a cat, supposedly requested by a client in a previous GET HTTP request.

Example: Complete HTTP Response

By assembling all defined response elements, a complete HTTP response would resemble the following structure:

```
HTTP/1.1 200 OK

Server:  Apache
Content-Type:  text/html
Content-Length:  235

<!DOCTYPE html>
<html>
<head>
    <title>Example Page</title>
</head>
<body>
    <h1>Hello, world!</h1>
    <p>This is an example page.</p>
</body>
</html>
```

In this example, the server replies to a sent GET HTTP request with a HTTP response, containing a `200 OK` status code, indicating a successful request. The response also returns the HTML content of a web page.

HTTP responses hold importance in this document, as they are extensively used in the developed program of this Master's Thesis. Indeed, in order to verify the validity of sent requests, the only formal indicator available is the server's HTTP response; Analyzing returned status codes is paramount, as it predominantly indicates the success or failure of a request.

2.1.5 Large Language Models

As Large Language Models are leveraged for the work of this Master's Thesis, a section is dedicated to their description. As Large Language Models can be interpreted as extensions of Language Models, the latter concept is first introduced.

Definition 2.7: Language Model

A **Language Model** is a computational model and a type of artificial intelligence that learns and predicts the probability distribution over a sequence of words by analyzing text data in a given language. Language Models are trained on numerous amounts of data, in order to recognize statistical patterns and semantic relationships found in given text data.

Definition based on: [50]

Essentially, Language Models consist of an algorithm that contains rules regarding the context of natural language. When data is fed to the algorithm, it will apply such rules in order to precisely estimate or produce new text content. Language Models are widely used in a lot of Natural Language Processing - NLP - domains such as machine translation, speech recognition, text summarization, dialogue generation and various other tasks related to text data.

Definition 2.8: Natural Language Processing

Natural Language Processing (NLP) is a subfield of computer science comprising of artificial intelligence and linguistics domains, focusing on the interaction between computers machines and human spoken language. Natural Language Processing involves the establishment of models enabling computers to understand human language in an accurate and meaningful manner.

Definition based on: [57] [81]

As the concept behind Language Models has been explained, Large Language Models can be formally defined.

Definition 2.9: Large Language Model

A **Large Language Model (LLM)** is a designation for Language Models generally encompassing neural networks containing numerous parameters and trained on extensive amounts of text data. Large Language Models render the understanding of complex language patterns and dependencies possible, and can accomplish a variety of different tasks.

Definition based on: [28] [51]

Considering that Large Language Models are trained on extensive datasets, such models are able to learn rich representations of language and usually generate coherent and contextually appropriate responses[3]. Moreover, in order to globally undestand the usage of a Neural Network, a definition is given.

Definition 2.10: Neural Network

A **Neural Network** is a computational model inspired by the biological neural networks found in the human brain. Neural Networks consist of interconnected artificial neurons, which are organized in layers. Neurons are given an input, perform a certain operation, and produces an output available for other neurons in the network. Certain neuron outputs can have more influence than other neuron outputs, which can be adjusted during training and is known as a *weight*.

Definition based on: [58] [79]

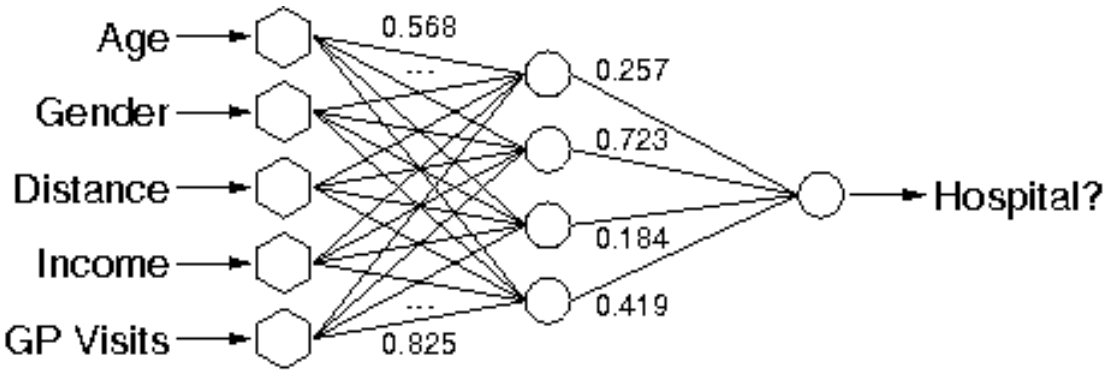


Figure 2: Example of a Neural Network, as given on the *Togaware* website [58].

Figure 2 displays a basic example of a Neural Network. As shown, initial input parameters are given on the left to neurons, which will calculate outputs to be given to neurons on the right. A final output is found on the foremost right side of the model, which represents the final output calculated. In this example, the Neural Network calculates if an individual should go to the hospital based on age, gender, income and other given

[3]However, as described in Section 3.1.3, generated responses can sometimes be incorrect as the model can *hallucinate*.

parameters.

As the focus of this Master's Thesis does not delve into a deep comprehension of Language Models and Neural Networks, preliminary definitions for a global understanding of such concepts have been given. Indeed, Language Models are not implemented nor modified in the Thesis, as the main point of interest is to analyze the performance and prompting of publicly available models for facilitating tasks in the computer science domain. Moreover, while a global understanding of publicly available models is possible, an in-depth understanding is not in reach as the relevant code is not open-source.

However, as *ChatGPT* and other models developped by the *OpenAI*[4] company are extensively used in the document, the following section is dedicated to describing the *ChatGPT* model.

2.1.6 The *ChatGPT* Model

ChatGPT is a chatbot technology that prominently rose in popularity shortly following its release in November 2022. It is recognized for its ability to engage in natural and coherent conversations, regarding a vast amount of different subjects and tasks. For instance, a user can task *ChatGPT* with 3 different prompts[5]:

- *"Explain quantum computing in simple terms."*
- *"Got any creative ideas for a 10 year old's birthday?"*
- *"How do I make an HTTP request in Javascript?"*

To which *ChatGPT* will give an adequate response, providing information regarding the prompted subject. In order to fully understand the underlying concepts of the chatbot, a formal definition is given.

Definition 2.11: *ChatGPT*

ChatGPT is a Large Language Model developed by *OpenAI*, designed for generating human-like text in a conversational environment. The name combines *Chat*, referring to the fact that it consists in a chatbot and *GPT*, referring to the Generative Pre-trained Transformer (GPT) architecture on which the model relies upon.

Definition based on: [39] [40]

Definition 2.12: Generative Pre-trained Transformer

A **Generative Pre-trained Transformer (GPT)** is a deep learning model architecture able to understand and generate human-like text. It combines two important concepts: **Generative Modeling**, consisting in creating new text, and **Transformers**, consisting in neural networks efficient at understanding language. GPT models are trained on a large amount of text data in order to learn patterns and structures in language.

Definition based on: [35]

The "pre-trained" aspect of Generative Pre-trained Transformer signifies that the model is initially trained on massive amounts of text data in order to learn new language patterns and structures. Through this pre-training step, GPT models are able to understand semantic relationships, context, and grammar similar to human language. Consequently, GPT models are effective in various NLP tasks including text generation, language translation and text classification. GPT models can also be fine-tuned in order to perform even more specific NLP tasks.

However, *ChatGPT* is not flawless, as Section 3.1.3 describes uncovered limitations commonly brought up and also discovered while using the tool.

[4] Not to be confused with the *OpenAPI* specification.
[5] The given examples consist in the 3 default examples given by *ChatGPT* when first launching its web interface [39].

2.2 API Testing

As briefly explained in the introduction, well designing, implementing and testing APIs is of foremost importance. Meticulously following this process greatly increases the reliability of APIs. The importance of this process is particularly reinforced for popular APIs which are regularly used by a lot of developers.

However, API developers are not completely isolated for this process. Regarding the API domain, there exists numerous tools that have been implemented, strengthening and alleviating the API development process.

The following sections will describe 2 existing API tools: ***Postman*** and ***RESTest***. *Postman* is used as an example to illustrate that the API lifecycle process can be alleviated, while *RESTest* gives an example of how APIs can be thoroughly tested. For each described tool, a general overview is given. Then, an example is given for illustration purposes. In addition, the performance of each tool is exhibited, along with a general summary.

2.2.1 *Postman*

Overview. *Postman* [65] is an API platform which provides developers with a collection of tools for designing, testing and managing APIs. *Postman* simplifies the API lifecycle process by offering a user-friendly interface and various useful features. For instance, developers are able to easily create requests, examine server responses or even collaborate with other team members through the platform. Key features of the *Postman* platform include:

- An **API client** to explore, debug and and test APIs. The client also allows developers to create complex API requests.
- The **design of API specifications** in various formats such as *OpenAPI*. *Postman* is also able to validate made specifications using a built-in engine.
- **API documentation** features, making the documentation a key part of the workflow. The generation of the documentation is not automatic in itself, but *Postman* is able to read markdown or machine documentation in order to automatically display it on a web page.
- An **API testing** feature, allowing developers to write various API tests. *Newman*, a command-line utility for *Postman*, allows the user to run the tests in a command prompt environment.
- **Mock servers** to simulate an API server in production and verify how real requests would be handled. Network latency can be simulated, by specifying custom request delays.
- **API monitoring** to have a visual overview of an API.
- An **API detection** tool, enabling users to capture network traffic.

As of April 2022, *Postman* had over 20 million users worldwide [25]. The home page of the *Postman* website states that this number is now over 25 million users [65], which in consequence brands *Postman* as one of the most popular API tool.

Example. As *Postman* is a complete API platform and not only an API testing tool, the following example comprises multiple figures containing illustrations of the described features.

As displayed in Figure 3, *Postman* enables developers to experiment with their APIs. In the upper center of the figure, users can generate requests and specify HTTP methods, parameters and headers. In the lower center of the figure, server HTTP responses can be observed, containing status codes, returned data and several other useful pieces of information. The right side of the figure displays the generation of a documentation relevant to the current route of the API, while the left side of the figure displays the API structure. In definitive, this specific examples illustrates the user-friendly interface and the ease of API handling granted by *Postman*.

Furthermore, the testing features of *Postman* can also be illustrated. Figure 4 displays an example of a test written on *Postman* and its corresponding interface. In this example, the test aims at verifying if the `gender` and `nat` parameter combination of a GET request result in correct response data for values `male` and `us` respectively. In order to do so, a test script is written. As the request specified that returned data should contain the `male` gender, the script also verifies that the `title` variable of returned data corresponds to a value of `"mr"`, representing the abbreviation of Mister. Then, the test script checks the value of the second parameter

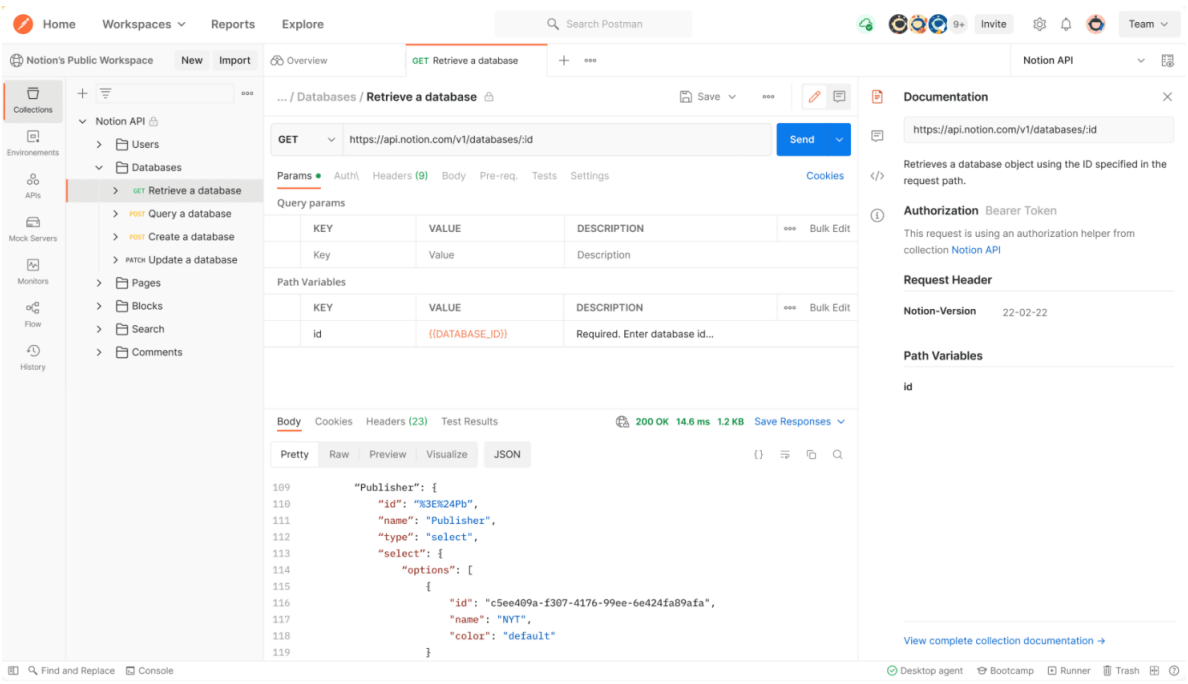


Figure 3: Example of the interface displayed by *Postman* [65].

`nat`. Below, test results can be observed, illustrating that each written test has successfully passed.

Performance. As *Postman* is a platform aimed at providing various tools assisting developers in API development, the testing feature is less performing than specific RESTful API testing tools. Indeed, while small tests can be easily written with *Postman*, other API testing tools are preferable such as *RESTest* which is presented in Section 2.2.2.

Overview: *Postman*

In summary, *Postman* offers a powerful and user-friendly interface for interacting with APIs, making it a very useful tool for API developers. *Postman*'s numerous tools and features enable efficient and reliable API integration and management.

2.2.2 *RESTest*

Overview. *RESTest* [71] is an open-source framework aimed at automating the process of RESTful API testing, in an entirely black-box-based approach. As no inspection of the source code is required, the tool can be used to test APIs written in any programming language. *RESTest* uses a model-based approach, which signifies that test cases are automatically derived from the given *OpenAPI* specification.

Definition 2.13: Black-box Testing

Black-box Testing is a testing technique where the internal structure and workings of a system under test are not known nor considered. Black-box Testing approaches rely entirely on observed program inputs and outputs to evaluate the system functionalities, based on the expected behavior and specifications.

Definition based on: [15] [38]

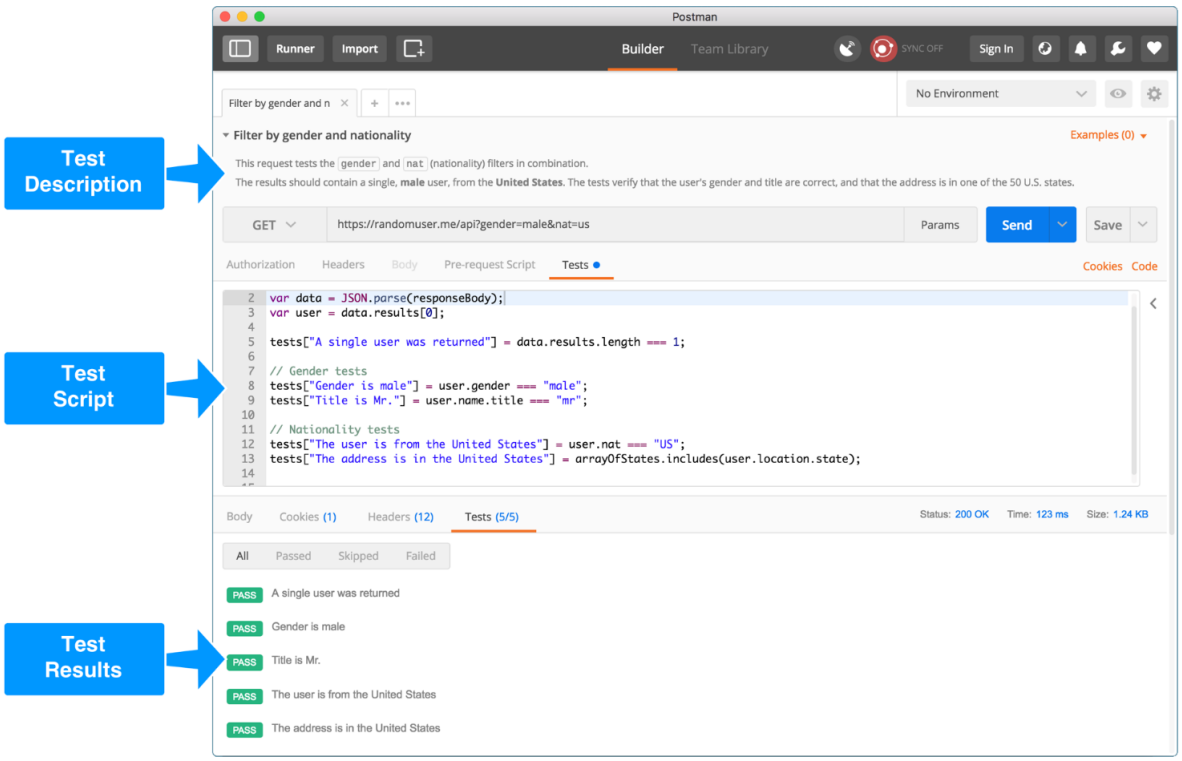


Figure 4: Example of a test written on *Postman*. The illustration can be found on a post of the *Blog* section of *Postman* [17].

Definition 2.14: *OpenAPI* Specification

The ***OpenAPI* Specification (OAS)** is a widely adopted industry standard used for describing the structure and behavior of RESTful APIs. The specification provides a structured scheme to define and document APIs. It provides details such as endpoints, request and response formats, parameters and authentication requirements. *OpenAPI* Specifications are usually represented in JSON or YAML formats. The following URL officially and thoroughly describes the OAS format: `https://github.com/OAI/OpenAPI-Specification`.

Definition based on: [61] [62]

RESTest can be interpreted as a testing *ecosystem*, as it provides a set of tools for automatically testing and monitoring APIs. Furthermore, *RESTest* diverges from customary API testing tools as it employs a series of robots. Indeed, in order to avoid repetitive test cases such as manually generating a request and asserting that the request exhibited a correct behavior, *RESTest* robots analyzes the *OpenAPI* specification of the API and distributes tasks to other robots in the ecosystem. The tool contains multiple categories of robots:

- **Test Bots** which generate and execute test cases. Test Bots are also sub-classified based on 3 elements:
  - **Write-safety**: Types of operations used by the bots.
  - **Test Data Generation Technique**: Technique used to generate the input data found in API requests.
  - **Test Case Generation Technique**: Technique used to build API requests, which represent the test cases.
- **Garbage Collectors** which delete resources created by previously described Test Bots.
- **Test Reporters** which generate test reports using the *Allure* [36] graphical framework.
- **Test Coverage Computers** which calculate the test coverage of the Test Bots based on coverage criteria of RESTful APIs.

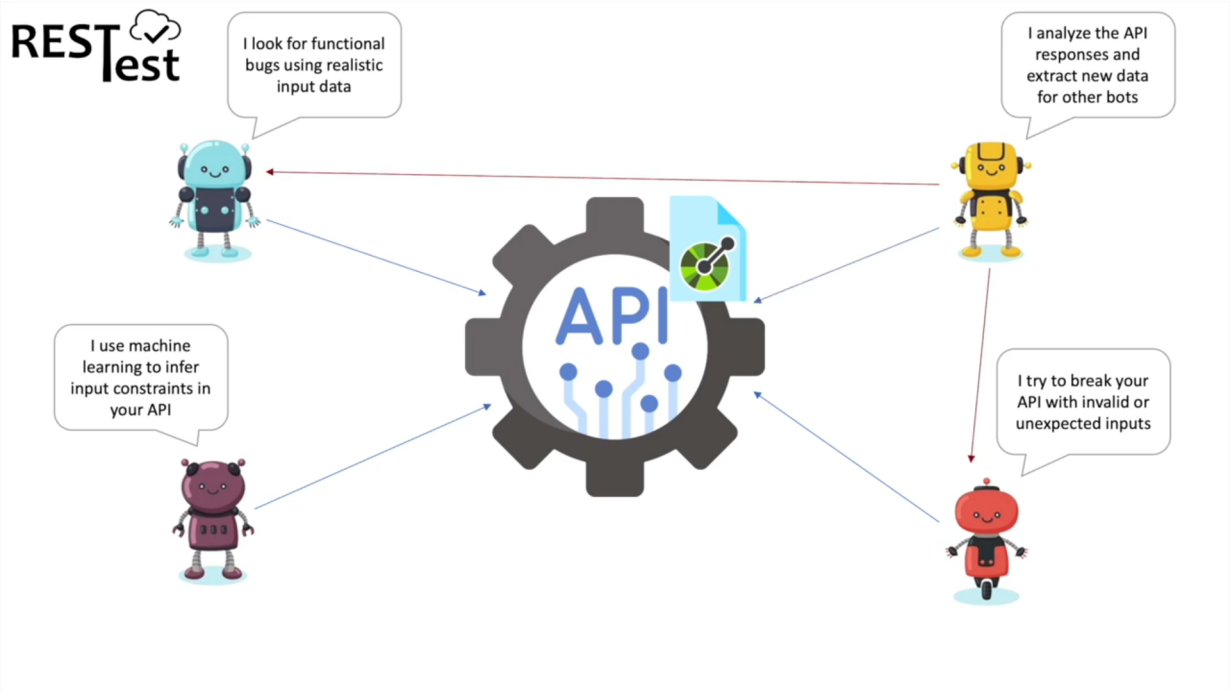


Figure 5: Example of a bot ecosystem of *RESTest*, as shown in the *YouTube* video presentation of the tool [22].

The interesting part about *RESTest* is that each Test Bot can contain various combinations of testing procedures. To illustrate, a Test Bot can have the following dimensions:

- **Write-safety**: *Read-only*, consisting in testing read operations such as HTTP GET requests.
- **Test Data Generation Technique**: *Data perturbation*, consisting in introducing small changes in the API request data in order to make it invalid.
- **Test Case Generation Technique**: *Random testing*, consisting in randomly assigning values to parameters.

Thus, the Test Bot given as example is labeled as R-DP-RT, combining all 3 dimensions into a unique test procedure. Figure 6 displays all possible dimensions for the Test Bots.

Bot dimension	Bot ID	Bot name	Description	Inclusion criterion	Total
Write-safety	R	*Read-only*	Tests only read operations (i.e., HTTP GET requests)	The API contains read operations	55
	RW	*Read/write*	Tests both read and write operations (e.g., POST and DELETE requests)	The API contains write operations	20
Test data generation technique	FD	*Fuzzing dictionaries*	Uses type-aware fuzzing dictionaries and malformed strings for all request parameters	All APIs	14
	CDG	*Customized data generators*	Parameters are configured with customized data generators (e.g., a generator of dates in 'yyyy/mm/dd' format)	All APIs	32
	DP	*Data perturbation*	Makes a minor change to the data used in an API request to make it invalid	All APIs	14
	SD	*Semantically-related data*	Uses concepts related to the semantics of the parameters, extracted from knowledge bases like DBpedia [64]	The API contains parameters from which to extract semantically-related concepts	5
	CD	*Contextual data*	Uses values observed in previous API responses	The API responses include reusable data for subsequent requests	10
Test case generation technique	RT	*Random testing*	Parameters are assigned values randomly	All APIs	39
	CBT	*Constraint-based testing*	Constraint solvers are used to satisfy all inter-parameter dependencies of the API [88]	The API contains inter-parameter dependencies	27
	ART*	*Adaptive random testing*	Similar to RT or CBT, but aiming to generate as diverse test cases as possible [82]	The API contains at least one operation with more than 10 parameters	9

*ART bots employ RT or CBT (depending on whether the API has inter-parameter dependencies [88] or not) as a basis to generate test case candidates, from which the most diverse test cases are selected [82].

Figure 6: Example of the *RESTest* Test Bot dimensions, as presented by Alberto Martin-Lopez et al. in the relevant paper [7].

Moreover, the *RESTest* ecosystem is described in a research paper entitled *"Online Testing of RESTful APIs: Promises and Challenges"* and is redacted by Alberto Martin-Lopez, Sergio Segura and Antonio Ruiz-Cortés [7]. A presentation video of *RESTest* can also be found on Alberto Martin-Lopez's *YouTube* channel [22].

Example. As *RESTest* enables automatic testing of RESTful APIs, an example of the user dashboard for monitoring purposes is presented in Figure 7

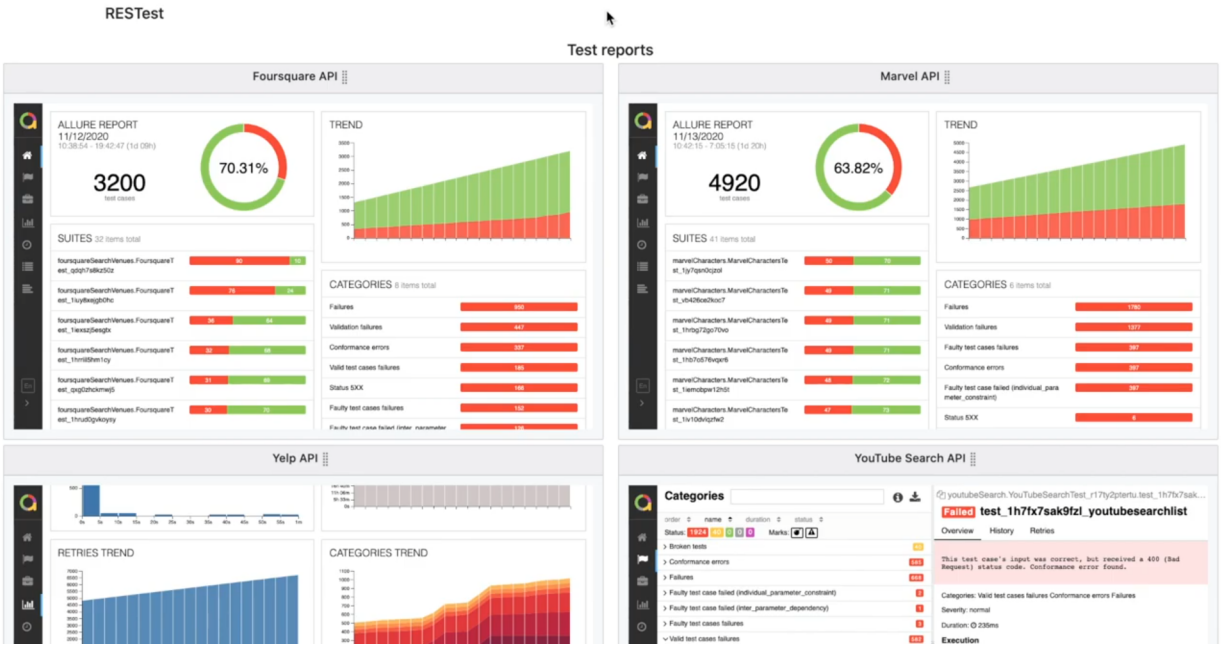


Figure 7: Example of the *RESTest* monitoring interface, as shown in the *YouTube* video presentation of the tool [20].

As displayed, the monitoring interface is capable of encompassing multiple APIs at the same time. Each monitored API has data displayed, ranging from report graphs - the upper 2 interfaces on the illustration - to in-depth error analysis - lower right interface on the illustration.

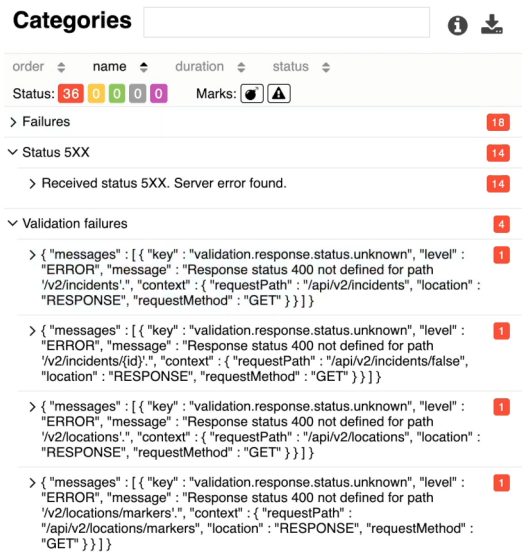


Figure 8: View of the *RESTest* interface displaying the found errors of an API, as shown in the *YouTube* video presentation of the tool [20].

Furthermore, Figure 8 illustrates a more in-depth view of an error interface. The types of obtained status codes are displayed, with drop-down menus for failures, 5xx status codes and validation failures. This type of interface allows users to monitor the validity of requests resulting from the testing process and analyze potential errors.

Performance. Performance-wise, developers of *RESTest* have experimented with the ecosystem for a month with over 10 existing RESTful APIs. Examples of such utilized APIs include the *Spotify*, *YouTube* and *Marvel* APIs. As certain APIs can be very large in size, not all API contents were tested. For instance, with the *YouTube* API, the developers tested the *Media* category with *Comments* and *Search* components as services under test. For the experiment, 100 bots were used. As a result, the bots were able to automatically generate 1,101,846 test cases, uncovering over 100,000 failures, resulting in more than 50 issues being found in all APIs under test.

Overview: *RESTest*

To conclude, the *RESTest* ecosystem allows developers to automatically test RESTful APIs by tasking various robots to do so. A user-friendly dashboard allows users to monitor the work of the robots in real time. The robots can run automatically for a predefined amount of time, which can be for multiple days, weeks or even months and even during the night, when the user is not in front of the computer. Thus, *RESTest* is a promising API testing tool displaying capable results.

2.3 Leveraging Large Language Models

2.3.1 Growth of the Technology

The use of Large Language Models in the software testing domain - and more prominently in the computer science domain - is particularly recent, since the favoured underlying technologies were only recently publicly available and gained popularity thereafter. *ChatGPT*, one of the most utilized LLM chatbot, was launched on November 20, 2022 [40]. *OpenAI Codex*[6], the LLM capable of generating computer code, was launched a year earlier, on August 31, 2021 [60].

Even though performing Large Language Models are particularly recent, their use is nonetheless fulminating in 2023 due to their incredible performance and task solving capabilities. For example, *TestPilot*, an adaptive test generation technique leveraging Large Language Models, was brought up in February 2023 [10] [74]. During the same month, a paper written by Aakash Ahmad et al. explored the use of *ChatGPT* in a human-bot collaborative software architecting [8].

However, before the launch of *OpenAI* models, researchers at *Google* introduced a family of language models in 2018 [2]. Their work is entitled *BERT*, acronym for Bidirectional Encoder Representations from Transformers. *BERT* inspired the research of other work, such as *CodeBERT* in 2020 [4]. More recently in 2022, Ahmed Khanfir et al. released *CodeBERT-nt*, a tool aimed at using pre-trained language models to infer code naturalness [6]. In 2023, Ahmed Khanfir et al. also researched efficient mutation testing via pre-trained language models [9]. As such work diverges from the leveraged models of *OpenAI*, they are only given as indicative basis for the existing work regarding language models and are not explained in detail.

2.3.2 Fuzz Testing via Large Language Models

On March 4, 2023, a scientific paper entitled *"Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models"* was published [11]. The paper, redacted by Yinlin Deng et al., explored the approach of fully automating the process of fuzzing Deep Learning libraries by leveraging the *OpenAI Codex* Large Language Model, and is to date the first research to do so. Their program, entitled *TitanFuzz*, achieved promising results, which consists in better code coverage than state-of-the-art fuzzing tools. For *TensorFlow*, an open-source software library for machine learning, *TitanFuzz* achieved 30.38% higher code coverage compared to state-of-the-art tools. For *PyTorch*, another machine learning framework, the same program achieved 50.84% higher coverage. Moreover, *TitanFuzz* detected 41 new bugs in the tested libraries, which were confirmed by the relevant developers to be previously unknown.

[6] *OpenAI Codex* is deprecated as of March 2023, as more general and recent models have replaced it.

Definition 2.15: Fuzzing

Fuzzing, also known as **Fuzz Testing**, is a software testing technique used to uncover errors or vulnerabilities in computer programs. Fuzzing consists in providing programs with random, invalid or malformed data as input, potentially triggering errors or unexpected behaviors.

The word fuzzing originates from a thunderstorm causing noise on a telephone line in 1988, in turn causing computer commands to get invalid inputs and crash.

Definition based on: [14] [16] (University Lectures) [34] [45] (Internet Sources)

As masking can refer to various different concepts, the following definition presents the utilized concept of masking.

Definition 2.16: Masking

Masking is a technique used to hide a piece of data from a set of data. Frequently, this piece of data is replaced with a *token*, a generic symbol applied onto the data in order to hide it. The token applied onto the data is called a *mask*.

Example: Masking Data

The following examples display different cases of masking a piece of data from a set of data:

- Masking a word of a sentence with a `<WORD>` token:
  `The cat is <WORD> home.`
- Masking a parameter of a method with a `<PARAM>` token:
  `result = execute(3, <PARAM>)`
- Masking a parameter value of a HTTP request with a `<VALUE>` token:
  `https://api.gbif.org/v1/species/?name=<VALUE>`

The idea behind *TitanFuzz* is simple: By generating code snippets utilizing the Deep Learning libraries and masking parts of these pieces of code, it becomes possible to ask the *OpenAI Codex* model to replace the masked elements in the code, thus generating new test values automatically. The code snippets are masked by using mutation operators, their role being to "mutate" certain sections of the code. Mutation operators can replace a method argument, a method name, etc. Figure 9 represents the algorithm utilized in the paper.

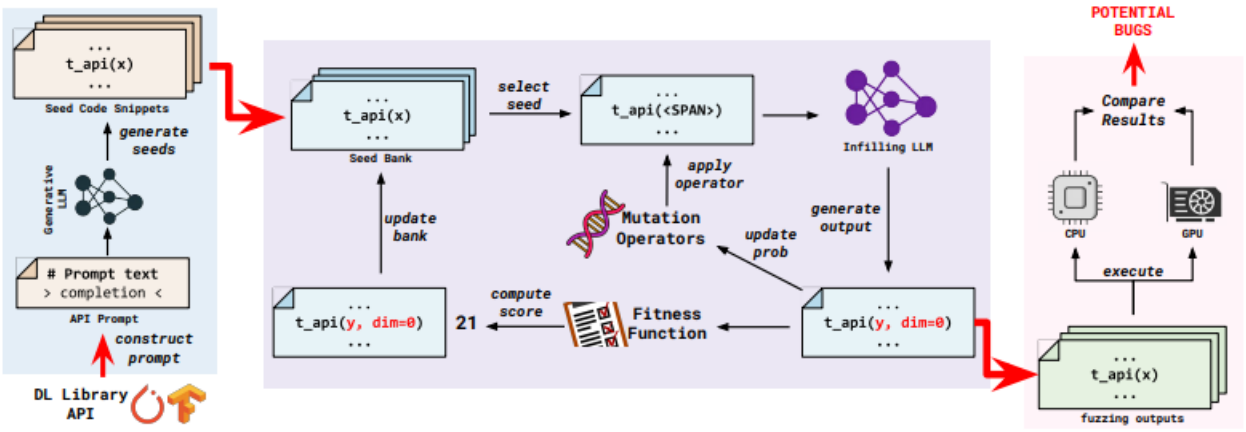


Figure 9: Overview of *TitanFuzz*, as presented by Yinlin Deng et al. in the relevant paper [11].

In order to fully understand the process, additional information is given. First of all, a seed bank is generated, initially containing seeds generated by the *OpenAI Codex* model. The seeds correspond to different code snippets, utilizing the Deep Learning library to be tested. *OpenAI Codex* is able to generate such seeds, thus automating the testing process even further. The distribution of each mutation operator is also initialized. This distribution will allow the selection of a preferable mutation operator for each mutation process, depending on the observed past performances of each mutation operator. Then, for each required iteration, a seed is chosen from the seed bank. Similarly to choosing a mutation operator, the chosen seed is selected based on a fitness score. This fitness score allows the selection of better code snippets, which supposedly utilize the Deep Learning library in a better and more extensive manner. When the fittest seed is selected, a mutation operator to be applied onto it is also chosen. As explained, the choice is not random, and is updated dynamically based on what works well with the current Deep Learning library. This learning process is calculated with a *Multi-Armed Bandit* algorithm.

Definition 2.17: Multi-Armed Bandit Algorithm

A **Multi-Armed Bandit (MAB) Algorithm** is an algorithm used in decision-making problems where an agent needs to choose between multiple options (arms) over a series of trials, the goal being to maximize cumulative rewards. It balances exploration - trying different arms to gather information - and exploitation - choosing arms with potentially higher rewards based on gathered information - in order to optimize the decision-making process.

Definition based on: [11] [53]

When an adequate mutation operator is chosen, it will then mask one or multiple pieces of code from the seed snippet, based on the mutation operator's implemented function. This mask consists in replacing the relevant code pieces with generic `<SPAN>` token tags. Once the masking process is finished, the masked code snippet is given as input prompt to the *OpenAI Codex* model. The model will then generate a new code snippet as output, which will be tested for potential bugs. Based on the latest code snippet, the mutation operator probability distribution is updated. The snippet will also be added to the seed bank with a computed score depending on the implemented fitness function. The entire process of selecting, masking, feeding and testing a seed is then repeated as desired, depending on the adequate number of test cases to be generated.

Definition 2.18: Mutation Operator

A **Mutation Operator** is an operator used to introduce variations into valid data in order to create slight variations of this data. Mutation operators can do so by randomly modifying 1 or multiple parameters of the given data. By introducing small changes into valid data, mutation operators can exercise different behaviors of the data, which can result in the discovery of new valid data.

For instance, mutation operators for a string can consist in inserting, deleting or bit-flipping characters of the string.

Definition based on: [16] [34] [46]

As of April 2023, Yinlin Deng et al. have submitted a research paper describing a new tool, *FuzzGPT*, enhancing the testing of Deep Learning libraries via Large Language Models. The paper is entitled *"Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT"* [12]. This research enhancement demonstrates that even though current work regarding Large Language Models exist, it continues to expand at a fast rate.

The process of masking elements with the help of mutation operators, and then feeding the result to a Large Language Model in order to obtain a new result is the core of *MutGPT*, the implemented program of this Master's Thesis. The contribution is not trivial; Such procedure is adapted for the purpose of generating, mutating and testing API requests to infer API specifications. More detail about the process is described in Section 4. Furthermore, the process is not to mask elements in order to test programs, but to adapt the process for the purpose of discovering - mining - API information.

3 Motivation

This section details the research that was carried out for the purpose of this Master's Thesis. First, the **Initial Research** is detailed, which concerns the discovery of the Large Language Model technologies that were utilized. Once familiar with the tools, the **Motivation** is presented, which details each strategy explored in order to efficiently leverage Large Language Models for the research. Finally, the **Request Mutation Strategy** is introduced.

3.1 Initial Research

Initially, the intention of this Master's Thesis was to explore new contributions regarding the API testing domain. Fortunately, the carried out research happened to take place at the same moment than the increase in popularity and performance of Large Language Model technologies. For this reason, initial experiments consisted in unearthing techniques that could allow such models to contribute to the API field.

Based on the documentation problem of APIs[7], an idea emerged. As Large Language Models embody and accumulate a plethora of data gathered from the Internet, what if it was possible to utilize this data in order to automatically generate and infer the documentation of APIs? As state-of-the-art research regarding this matter has not been explored, leveraging Large Language Models to automatically infer API specifications consists in the baseline of this Master's Thesis, and is researched thoroughly.

3.1.1 Discovery of *ChatGPT*

To begin, initial experiments were required in order to discover how Large Language Models function. Accordingly, experimentation began with the use of the most popular LLM at the time, which was - and still is as of today - *ChatGPT*.

ChatGPT was first used through its web interface, available at the following URL: `https://chat.openai.com`. In order to use the model, the creation of an *OpenAI* account is required. A valid phone number is notably needed, along with an email address. The web interface allows the user to input a message as prompt, and receive an answer from the *ChatGPT* model accordingly. Moreover, the user can engage in conversations with the model, the latter remembering previous prompts of the user. Figure 10 presents the interface of *ChatGPT*.

The *OpenAI* website also offers a service entitled *Playground*, where web users are able to experiment with the prompting of various models. It is also possible to modify numerous parameter values, influencing the responses of the model. Figure 11 displays an example of the *Playground* web page.

3.1.2 *OpenAI Python* Library

Using *ChatGPT* through a website interface is not ideal to automatically generate API documentations, as the user would need to manually go to the web page, input individual prompts and capture the relevant answers. Thankfully, there exists an official *OpenAI Python* binding, allowing developers to construct in-code HTTP requests to the available *OpenAI* models. The API reference for the *OpenAI* library is available at the following URL: `https://platform.openai.com/docs/api-reference`.

To utilize the *OpenAI Python* library, an API key is required in order to authenticate users making HTTP requests to the *OpenAI* server. The key can be generated by users disposing of an *OpenAI* account, at the following URL: `https://platform.openai.com/account/api-keys`. If required, a user can generate multiple API keys.

The *OpenAI Python* library allows developers to create *completion requests*. Such requests specify a model to be used, a message to be prompted to the model and other parameters impacting the model response. Listing 1 illustrates an example of a completion request in *Python*. As displayed, the *OpenAI* request contains the following parameters:

- `engine`: The *OpenAI* model to be used. In this case, it is the `text-davinci-003` model of the class *GPT-3.5*.

[7]The API documentation problem refers to the fact that API documentations are often lacking, incomplete or informal, as described in Section 1 and Section 2.

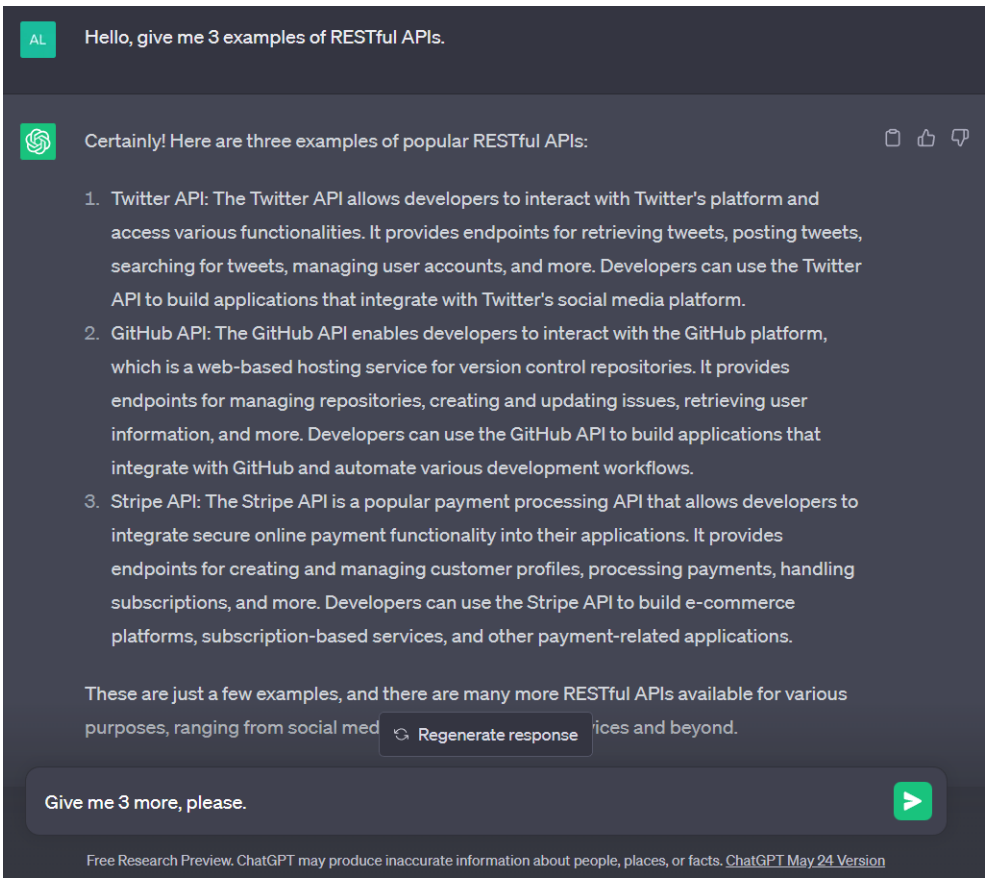


Figure 10: Interface of *ChatGPT*, with an ongoing conversation.

```
request = openai.Completion.create(
                engine = "text-davinci-003",
                prompt = "Give me a definition of an API.",
                max_tokens = 1024,
                temperature = 0.7
                )
```

Listing 1: Example of a *completion request* for the *OpenAI* library in the *Python* programming language.

- `prompt`: The text to be given to the model, representing the required task. In the example, the model is prompted to give a definition of an API.
- `max_tokens`: The maximum amount of tokens that the response can contain. The *OpenAI Tokenizer* [76] page states that on average, 1 token corresponds to 4 characters of text in English. In this case, the model response should contain a maximum of 1024 tokens, representing about 4096 characters.
- `temperature`: The randomness of generated responses, between 0 and 2. A lower temperature will generate more deterministic outputs, while a higher temperature will make outputs more random. In this case, the temperature is set to 0.7, which is the default temperature utilized by *ChatGPT*.

As of May 15, 2023[8], there exists 8 different models available in the *OpenAI* library. For the research experiment and the implemented tool, only 2 of these models were employed: *text-davinci-003* and *gpt-3.5-turbo*.

[8]The mention of the exact date of the number of available *OpenAI* models is important, due to the fact that since the beginning of this Master's Thesis, GPT 3.5 and GPT 4 models have been released, completely revamping and enhancing model capabilities. Thus, it is quite possible that in a near future, more performing models could be released.

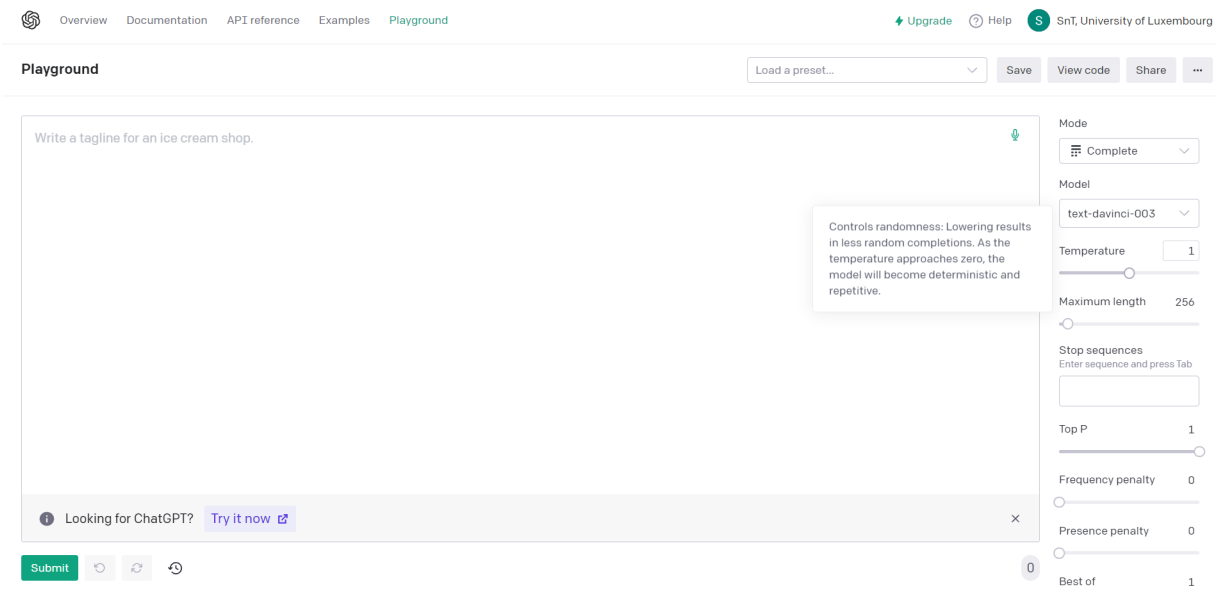


Figure 11: Interface of the *OpenAI Playground* feature. The figure notably displays a *temperature* slider bar, allowing the user to modify the randomness of model responses.

3.1.3 Limitations

As experimentation went on, a couple of difficulties and limitations were encountered regarding the prompting of *OpenAI* models. Such drawbacks are presented in the following paragraphs. All sources regarding model pricing and token limiting were found on the *Models* page of the *OpenAI* website [52], as of May 2023.

Model Pricing. First of all, prompting *OpenAI* models is not free of charge. Each model comes with a specific pricing tag, depending on the amount of tokens returned in given responses. For instance, the pricing of the *gpt-3.5-turbo* model is $0.002 per 1000 tokens as of May 2023. When creating an *OpenAI* account, a certain amount of credit is granted for a free trial usage. The amount of credit corresponds to $18 - later reduced to $5 for new *OpenAI* accounts. However, the strategies described in the following sections require a lot of prompts in order to function, which swiftly make use of the free trial credit. Figure 12 illustrates an example of the *OpenAI* API usage page. In this specific example, the API usage for the month of March 2023 is presented. The daily usage along with the free trial credit is displayed. Furthermore, a breakdown of model usage is also accurately shown, detailing the time at which requests were made, to which model and with the amount of tokens used.

Response Token Limit. Depending on the model used, each response returned contains a maximum amount of tokens. For the *gpt-3.5-turbo* model, the maximum amount is 4,096 tokens. This signifies that if a given response exceeds the maximum token limit, it is possible for it to abruptly terminate. Thus, a user can receive an incomplete response and without any warning.

Varying Response Times. As the *OpenAI* API is freely available to all - with a free trial at least, the server endpoint can occasionally receive a lot of load from sent requests. Thus, the response time when prompting a model can become inconsistent and vary depending on the server load. Some of the strategies presented in the following sections require a lot of requests to be sent one after the other, which consequently need fast response times.

Rapidly Growing Technology. As this Master's Thesis went on, *OpenAI* has constantly updated and improved its API. In February 2023, the best available model on the API was *text-davinci-003*, with a response token limit of 2,000 at the time. As of May 2023, the API contains over 8 classes of various models. The latest class currently in limited beta, *GPT-4*, contains models even more capable than the previous *GPT-3.5* class and with a more recent training data, as stated on the *OpenAI* website. Moreover, the new *gpt-4-32k* model has an immense token limit of 32,768, thus rendering the response token limit problem practically obsolete. Since this recent technology was uncovered lately during the production of this Master's Thesis, it is not considered but it remains interesting for prospective work.

Hallucinations. Large Language Models are often described as "confidently incorrect", or in more popular

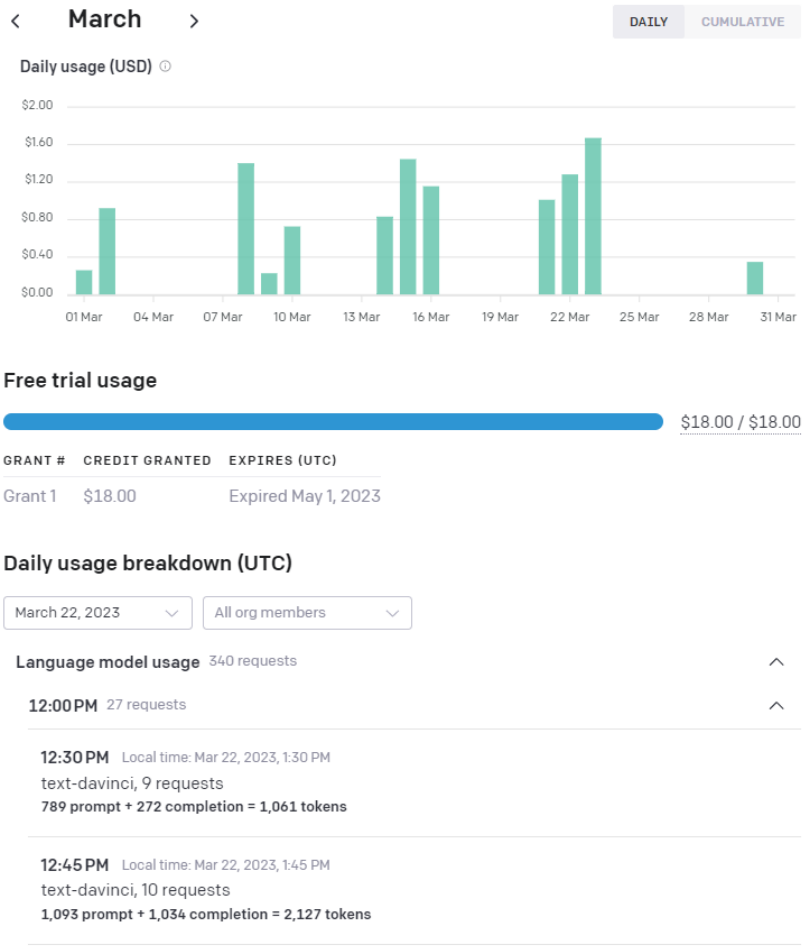


Figure 12: Example of the *OpenAI* API usage page for a user.

terms, *hallucinating*. This ascertainment is due to the fact that such models will, in certain cases, give an answer that seems convincing and well detailed, but is in fact false with regard to the given prompt.

Definition 3.1: Large Language Model Hallucination

A **Hallucination** in the Large Language Model domain refers to a phenomenon where a model generates text that may appear coherent and contextually valid but contains inaccuracies, inconsistencies, or unrealistic information. Such responses can be disconnected from the real world and based on false assumptions.

Definition based on: [29]

A popular example of such behavior is asking *ChatGPT* if chicken eggs are larger than cow eggs - cf. Figure 13. While this question might seem foolish for a human being, it is interpreted in a completely different manner by the Large Language Model. The given example has to date been patched, but a similar problem still remains nonetheless.

3.2 Preliminary Findings

Having acknowledged the initial research progress along with the encountered difficulties, the explored strategies aimed at automatically generating API documentations are now presented. For each strategy, a formal prompt with the utilized *OpenAI* model is given, along with encountered difficulties.

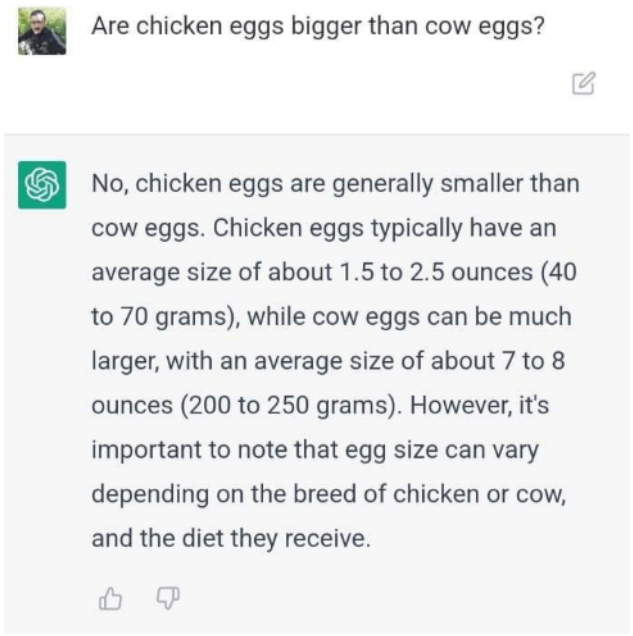


Figure 13: Example of a "hallucination" displayed by *ChatGPT* when asking if chicken eggs are larger than cow eggs.

3.2.1 Initial Strategy: Asking the Model for API Documentations

To begin, a starting idea that comes naturally to mind is to simply prompt the model and ask it to generate the documentation of an API. If the model is able to generate a full specification, then the objective is achieved. Since API documentations can be formally defined in an *OpenAPI* specification, an approach would consist in simply prompting the model with the following task:

Prompt: Direct Specification Prompt Strategy

"Generate the OpenAPI specification of the `API_NAME` web service."

Parameter:

- `API_NAME`: Name of the API.

Leveraged Model: *ChatGPT*

The described approach will be labeled as the **Direct Specification Prompt Strategy (DSPS)** in the following paragraphs. Since the approach is quite trivial in terms of automation, the *ChatGPT* web interface will be used to display the obtained results.

To illustrate the strategy, the *GBIF Species* API is chosen as an example. To briefly elucidate, *GBIF* stands for *Global Biodiversity Information Facility*. This information facility is an international network database, and its *Species* API provides access to data regarding numerous species existing around the world. The *GBIF Species* API contains various routes and parameters in order to retrieve such information in GET requests.

By following the prompt scheme described in the previous paragraphs, the following task is given to the model: *"Generate the OpenAPI specification of the GBIF Species API web service"*. Figure 15 displays the response generated by the Large Language Model. As shown, *ChatGPT* is able to answer with an *OpenAPI* specification structure, containing information regarding the prompted API.

Problem 1: Token Limit. However, the response abruptly interrupts in the middle of the *OpenAPI* specification, as displayed in Figure 16. This inconvenience is due to the fact that outputs generated by the model are limited to 4,096 tokens. Thus, generating a complete specification is not possible, since it will oftentimes

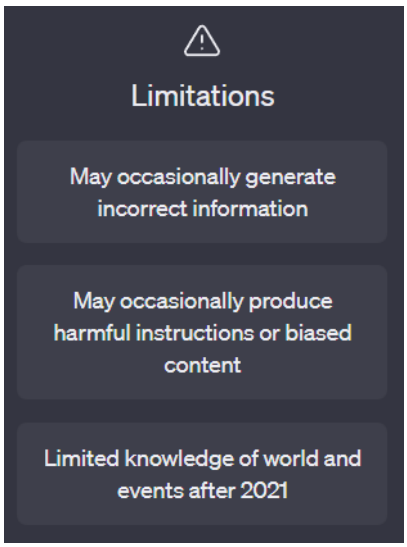


Figure 14: Overview of *ChatGPT*'s limitations, as described on the *OpenAI* web page [39].

exceed the token limit for the average API. As mentioned in Section 3.1.3, the rapidly growing aspect of Large Language Model technologies could render this problem obsolete. Perhaps, the *gpt-4-32k* model having over 32,000 tokens as limit could feasibly generate complete *OpenAPI* specifications. Due to the fact that such technology was not available during the overall research of this Master's Thesis, it is consequently not considered. Moreover, for very large APIs containing various routes requiring detailed specifications, such model could not generate a complete structure.

Problem 2: Confidently Incorrect Specifications. The "confidently incorrect" behavior of Large Language Models detailed in Section 3.1.3 could also pose a problem to the DSPS. For instance, an *OpenAPI* specification given by *ChatGPT* could contain an invalid parameter, which would seem valid but would cause issues when generating requests based on the specification grammar. Additionally, detecting an invalid parameter in an API grammar is not an easy task, as one would need to generate multiple different requests containing the invalid parameter to analyze its validity in a more in-depth manner. Furthermore, certain valid parameters may appear invalid when inserted in requests on their own, as they would need to be paired with other parameters in order to exhibit a valid behavior.

Overview: Direct Specification Prompt Strategy

While the Direct Specification Prompt Strategy seems easily automatable and straightforward for very small APIs, it does fall short for a various number of explained reasons. The token limit of model responses is a first limitation. Moreover, acquiring a complete specification of an API from a model is overwhelming, as each element of the grammar is susceptible to the "confidently incorrect" behaviour of LLMs. In consequence, this strategy is not explored further and leaves room for different strategies in the following sections.

3.2.2 Change of Direction: Asking the Model to Generate Requests

As the initial strategy proved to be underwhelming, other approaches needed to be considered. The Direct Specification Prompt Strategy uncovered 2 underlying problems that necessitate solutions. First of all, to shelter against the "confidently incorrect" behavior of models, a certain course of action needs to be found in order to easily verify if a response generated from a model is indeed incorrect. Second of all, responses generated by the leveraged models cannot exceed their token limit, to avoid sudden and troublesome output interruptions. By combining these 2 statements, an appealing solution exists: asking the model to generate HTTP requests.

Indeed, directly receiving API HTTP requests as output would solve the stated issues. Regarding the "confidently incorrect" behavior, a request is either valid or invalid - there is no in between. Thus, by asking the model to generate HTTP requests, it is easy to verify the validity of such request by sending it to the appropriate

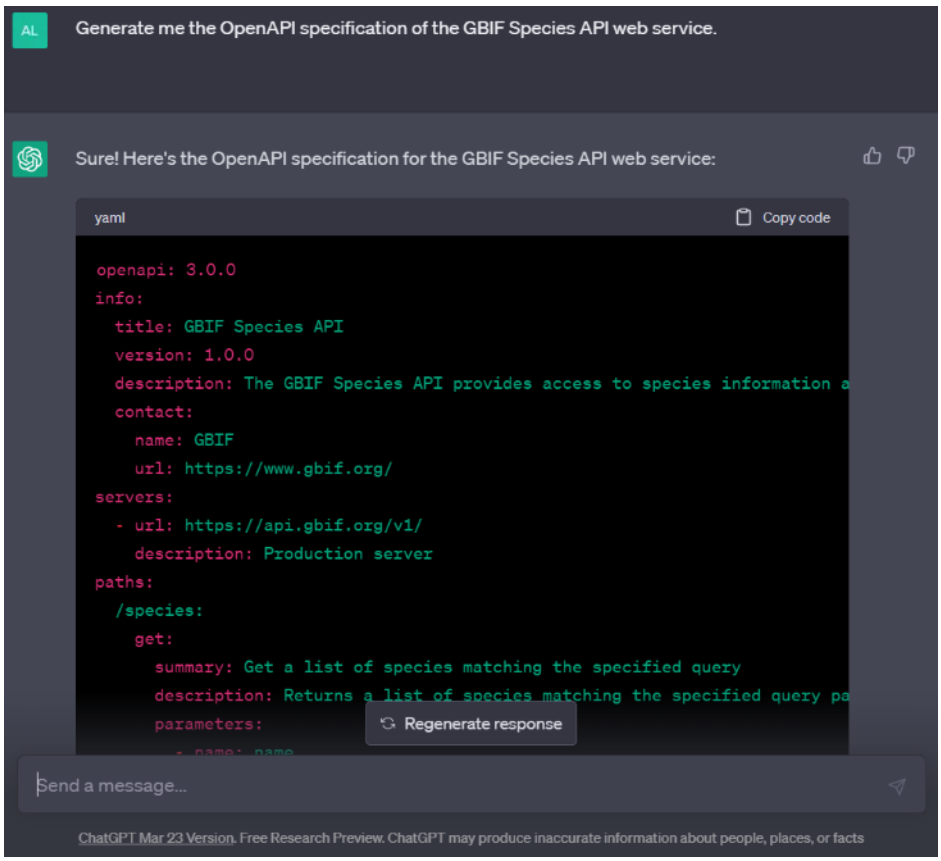


Figure 15: Example of a response generated by *ChatGPT* for an *OpenAPI* specification of the *GBIF Species* API.

API endpoint. As for the response token limit, the average request rarely exceeds 100 tokens[9]. The token limit would in consequence not be attained, which would solve the second underlying issue.

In order to achieve the initial goal of automatically generating an API documentation, the generated valid requests can be decomposed for the purpose of collecting information about the API, containing routes, parameter names and parameter values. The objective of such strategy would be to ask the model to generate various valid requests, each request covering different routes and parameters of the API, thus automatically inferring the specification of the API.

A first strategy consists in asking the Large Language Model to generate various valid requests. A basic prompt for such task consists in the following:

[9]Result obtained by calculating the average amount of tokens used for 10 different requests with the *OpenAI Tokenizer* feature [76].

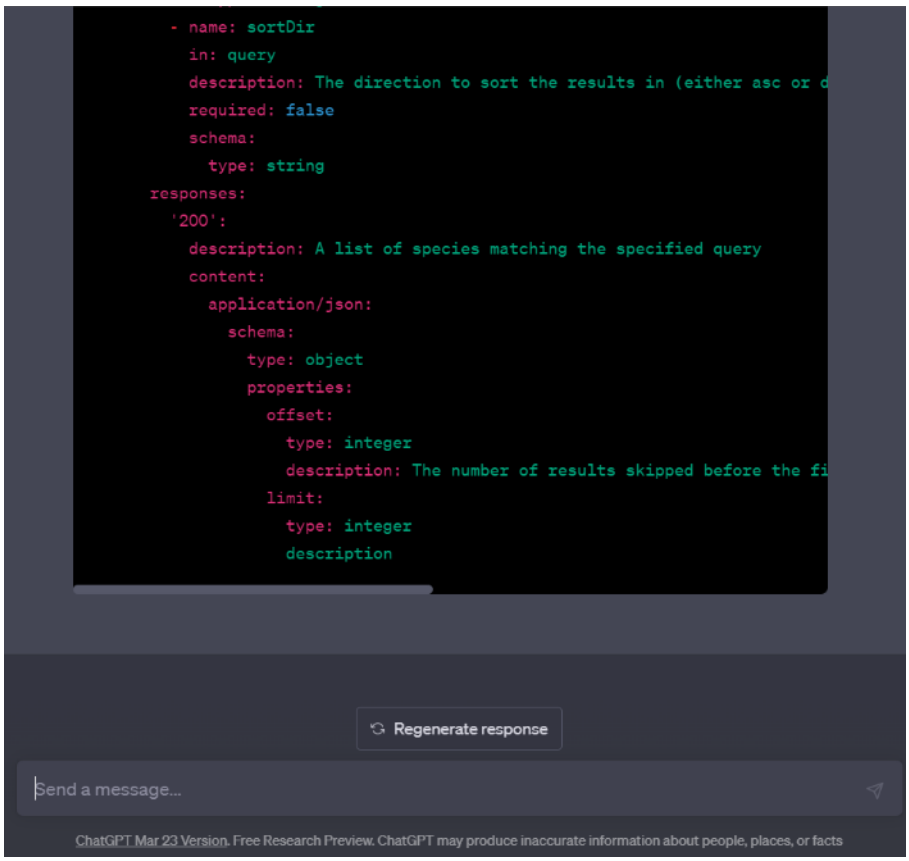


Figure 16: Example of a *ChatGPT* response that abruptly interrupts at a `description` parameter name when the 4,096 token limit threshold is attained.

Prompt: Basic Request Prompt Strategy

"Generate `X` examples of GET HTTP requests that can be made to the `API_NAME` web service."

If an API key is required, add: "The API key to be used in requests is the following: `API_KEY`."

Parameters:

- `X`: Number of required requests.
- `API_NAME`: Name of the API.
- `API_KEY`: Key to be specified in API requests, if required.

Leveraged Model: *text-davinci-003*

Additionally, if the API requires an API key in order to authenticate requests, it is possible to augment the prompt with a phrase describing the API key to be used. The exact key should replace the `API_KEY` placeholder in the detailed prompt. As RESTful APIs mainly distribute representations of specified resources, the prompt specifies that the generated HTTP requests should be of the appropriate HTTP method, which is the GET method. The current approach is designated as the **Basic Request Prompt Strategy (BRPS)**.

By experimenting with the current strategy, the *text-davinci-003* model proved to correctly generate examples of GET HTTP request for the specified API. However, by manually analyzing the generated requests, various issues regarding the strategy were uncovered.

Problem 1: Inconsistent Response Structures. When prompting the model for examples of requests, it is possible for the generated responses to vary in their structures. Figure 17 exhibits an example of such structure difference found between 2 different responses of the model, both answering to the same input prompt.

While the structure difference is not a problem for the understanding of a human being, it greatly hinders a machine's comprehension. Undoubtedly, in order for a computer program to analyze HTTP requests, such requests should be formally returned.

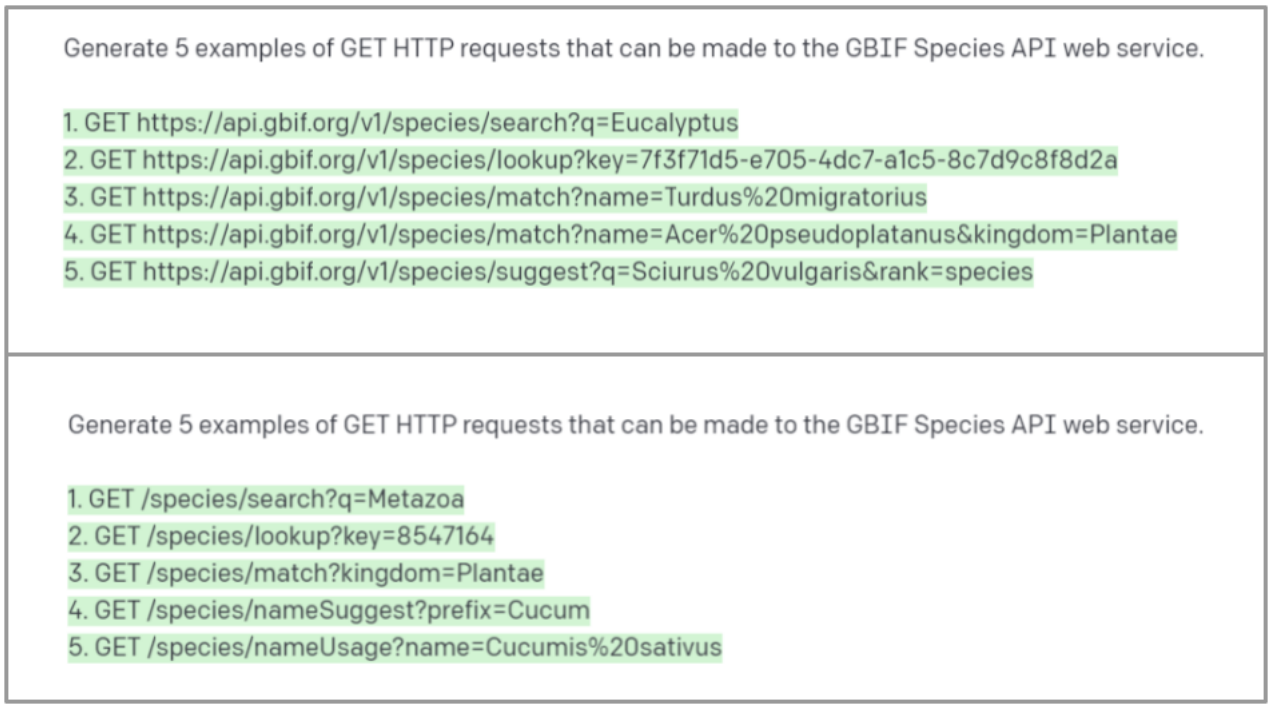


Figure 17: Example of 2 different response structures given by the *text-davinci-003* model for the same prompt.

In spite of such issue, a feasible solution consists in prompting the model with a supplementary task:

"The example requests need to be in a Python list of the following structure: `["request 1", "request 2", ...]`."

By supplementing the prompt with this statement, the model is able to generate a response containing the described data structure for requests. Even though it is possible for the response to contain additional information regarding the generated requests, parsing the response text allows the program to recover the response section containing the queried *Python* list. Figure 18 displays an example of the returned response in the specified structured.

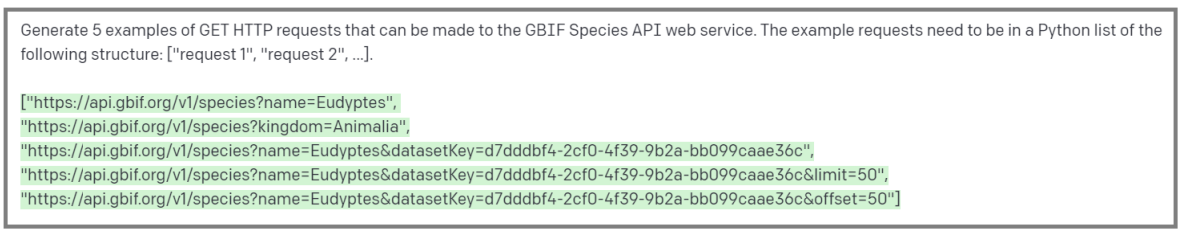


Figure 18: Example of the *text-davinci-003* model returning a response in a *Python* list, as specified in the prompt.

Problem 2: Inconsistent Validity and Variety of Requests. While the generation of requests is achievable with the ongoing strategy, obtained results were found to be lacking in terms of validity and variety. A formal example of an experiment with the prompting of 20 requests for 4 different APIs is described in Table

3. As Section 5.3 already evaluates and compares the performance of each strategy, the obtained results are given at present on an indicative basis.

In order to elucidate, the strategy is indeed capable of generating HTTP requests to the given APIs. However, responses returned by the model are inconsistent and unreliable. Another experiment[10] was carried out, prompting the model to generate 20 requests for the *GBIF Species* API. For this result, 17 of these requests proved to be valid, each containing a different route. At first glance, the result seems sufficient. However, further analysis displayed that the requests are unvaried. Indeed, the model generated routes such as `/species/1, /species/2, /species/3`, each with a different number corresponding to the ID of a specific species. While these routes are valid for the given API, their diversity is lacking. Other examples of such inconsistencies include:

- When asking for 10 requests to the *MusicBrainz* API, all generated requests were invalid. The process was repeated 2 times to check if valid requests could still be generated, which proved to still result in invalid requests.
- When asking for 20 requests to the *MusicBrainz* API, the model was able to generate all 20 requests. However, each request was missing the base route of the API, which in consequence rendered such requests invalid. The same result was also observed when asking for 20 requests to the *Helioviewer* API, where all generated requests were missing the base API route.

Overview: Basic Request Prompt Strategy

The Basic Request Prompt Strategy proved to be able to generate valid and structured requests, for prompts specifying a number of requests to be generated for an API. While the observed results are lacking in terms of consistency and validity, this simple strategy is certainly a step in the right direction in order to generate API documentations. Assuredly, the approach requires further investigation.

3.2.3 Improving the Request Generation Strategy

In order to improve the previous strategy, the underlined problems need a solution. The first problem regarding the inconsistent response structure was solved by supplementing the input prompt with a new assignment for the model, detailing that it should return requests in a *Python* list data structure. As this solution proved to be performing, it is kept for the improvement of the previous request generation strategy. Regarding the second problem, a new solution needs to be found in order to improve the consistency of generated requests.

By experimenting with different input prompts, a functioning solution was found. The idea consists in complementing the BRPS input prompt with various details, in order for the model to generate complete, varied and valid HTTP requests for the specified API.

[10]The experiment is not formally described in a table in this document, as the explanation is sufficient in order to demonstrate the strategy results.

Prompt: Complex Request Prompt Strategy

"Generate `X` examples of complex GET HTTP requests that can be made to the `API_NAME` web service. The example requests need to be in a Python list of the following structure: `["request 1", "request 2", ...]`. The requests should all be valid, have various routes and different parameters. The requests cannot be too similar. Do not forget to include the base route of the API in each request."

If an API key is required, add: "The API key to be used in requests is the following: `API_KEY`."

Parameters:

- `X`: Number of required requests.
- `API_NAME`: Name of the API.
- `API_KEY`: Key to be specified in API requests, if required.

Leveraged Models: *text-davinci-003* and *gpt-3.5-turbo*

As described, the new input prompt contains more information regarding the requests to be generated by the model. The prompt mentions that the requests need to be varied, containing different routes and parameters. Similarity is also mentioned, in order to warn the model that the generated requests cannot be too identical. Ultimately, the inclusion of the API base route in the requests is also mentioned, to avoid incomplete request structures. The improved request generation strategy is entitled the **Complex Request Prompt Strategy (CRPS)**.

In Section 5.3.3, Table 4 presents the results obtained with the strategy, for the exact same experimentation as the BRPS. The *gpt-3.5-turbo* model was initially used, but then replaced by the *text-davinci-003* model as obtained results from both models were principally identical in terms of generating requests. The replacement is due to the fact that the *text-davinci-003* model is less costly in terms of tokens[11]. Moreover, the *gpt-3.5-turbo* model occasionally did not return requests when prompted to do so, but returned the following phrase:

"As an AI language model, I don't have real-time access to the internet or the ability to browse websites. My responses are based on the information available to me up until September 2021. Therefore, I cannot provide you with..."

Thus, the *text-davinci-003* model was preferred, as such responses were not generated even though the model's training data limit is similar.

By analyzing outputs, the Complex Request Prompt Strategy obtained better results than the Basic Request Prompt Strategy. The amount of valid requests is greater, and less invalid and unvaried requests are observed. Supplementing the input prompt with additional details proved to be effective, as the generated responses contain an increased number of different routes and parameters. As explained in the previous strategy, Section 5.3 already evaluates and compares the performance of each strategy; The obtained results are given at present on an indicative basis only. However, a couple of lesser problems were uncovered.

Problem 1: API Related Request Formats. A couple of responses with the Complex Request Prompt Strategy displayed invalid request formats, relative to the utilized API. For instance, a small-scaled experimentation[12] with the *Helioviewer* API [47] generated requests which appeared to be valid. However, with further investigation, each request is missing a `"/"` character before the description of the parameters. For instance, the request:

`https://api.helioviewer.org/v2/getJP2Image?date=2014-01-01T23:59:59Z&sourceId=14`

[11]The model costed less as of March 2023. It is possible that in the future, *OpenAI* pricing change depending on the model.
[12]The experimentation is not formally given in the document.

Is invalid for the *Helioviewer* API, whilst the request:

`https://api.helioviewer.org/v2/getJP2Image/?date=2014-01-01T23:59:59Z&sourceId=14`

Is valid, since it contains the `"/"` character before the parameters. While discarding the `"/"` character before the parameter description is considered to be permissible for modern browser standards [18], it may not always result in valid requests, as displayed in the current example. Due to the fact that the *text-davinci-003* model is trained on modern day Internet data up to 2021, it probably omitted the `"/"` character as most modern API requests from the pertinent training data do not include it.

Even though such problem is quite limited, it remains important to consider its probability of occurrence. A straightforward solution consists in always adding a `"/"` character before the request parameter description, as it always results in valid requests, as long as the request is by all means valid. Thus, when acquiring generated requests from the model, the program will parse the requests and simply add a `"/"` character at the adequate position.

Problem 2: Greater Model Response Time. By enhancing the input prompt, the model response time increases. This is due to the fact that the model needs to generate more specific and varied requests, resulting in a longer time to return the response. The issue is further amplified when prompting the model for a large amount of different requests. By pairing this issue with the unpredictable *OpenAI* server load, the waiting time for the response of a CRPS prompt can be lingering. To illustrate, a CRPS prompt for only 20 requests of the *REST Countries* API [70] resulted in a model response time of 99.62 seconds[13].

Overview: Complex Request Prompt Strategy

While the Complex Request Prompt Strategy improved the Basic Request Prompt Strategy in terms of request validity and variety, an increase in the response time of the model causes issues with the generation of numerous requests. Such issue hinders the API specification inference objective, as large APIs would necessitate a lot of different requests from the model in order to properly infer a sufficiently complete documentation.

3.3 Request Mutation Strategy

In order to improve preliminary strategies for the purpose of automatically generating API documentations, the process of masking and mutating requests with Large Language Models was explored, comparably to the work presented in Section 2.3.2. To recall, it is possible to hide a piece of data from a data set, in order for the model to replace the hidden piece of data with known training data. Thus, the training data of the *text-davinci-003* model could be leveraged to generate new valid requests.

Preliminary experiments with the *text-davinci-003* model demonstrated that masking a part of an example request with an `<op>` token mask generated in fact a new request. Figure 19 exhibits an example of the model returning a new valid parameter `offset=0` when prompted to replace a mask, hiding a parameter of a request to the *GBIF Species* API.

As the request masking and mutation approach proved to be viable, a tool entitled *MutGPT* was developed for the Master's Thesis, expanding from this strategy. The tool is presented in the following section and consists in the main contribution of the research.

[13]Result obtained in March 2023, with a stable Internet connection. However, Section 5.3 describes that obtained results for the CRPS do not exceed a minute. As evaluations for RQ2 took place in May 2023, *OpenAI* servers could have improved or server load could have been less important at the day of the experimentation.

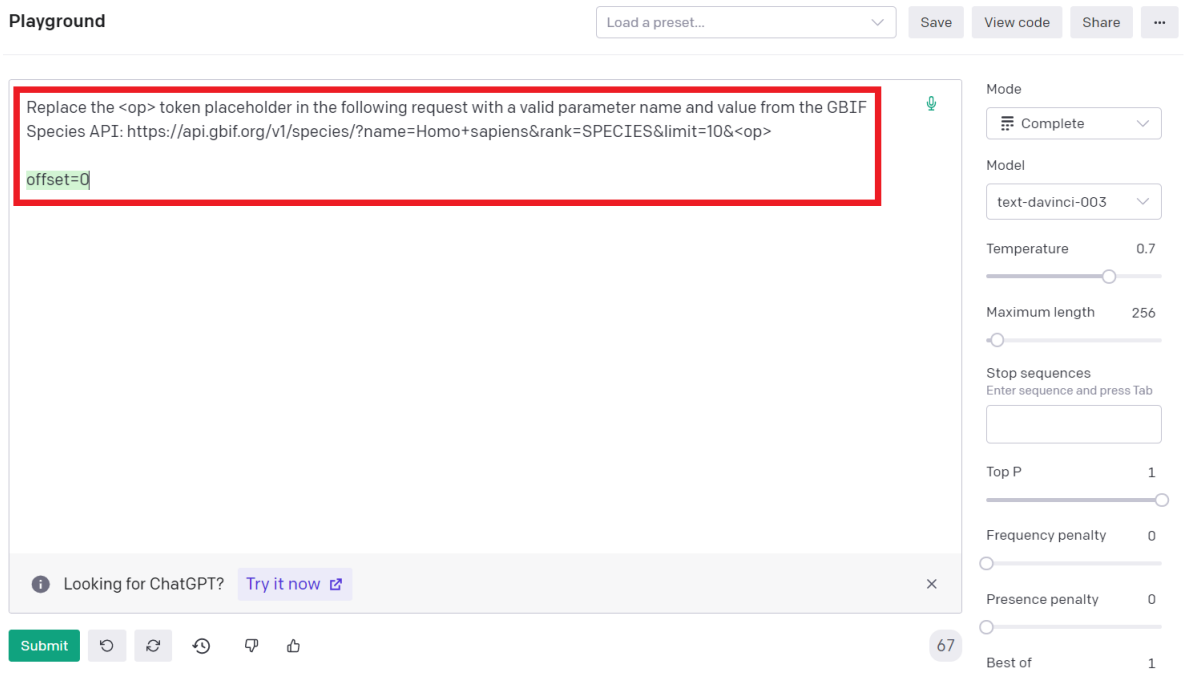


Figure 19: Example of a response given by the *text-davinci-003* model when asked to replace a masked parameter in a request. The interface is the *Playground* [64] web feature of *OpenAI*.

4 *MutGPT*: The Automatic Request Mutation Tool

This section is dedicated to the description of *MutGPT*, the tool developed for this Master's Thesis in order to **(1)** automatically generate and mutate RESTful API requests and **(2)** infer the specification of such APIs by leveraging Large Language Models. The tool was assembled following the various strategies explored throughout Section 3.2.

First, a preliminary **Overview** of the tool is presented. Next, in order to fully grasp the methodology utilized in the program, the **Mutation Operators** and the **Request Validity Verification** sections are detailed. Afterwards, a more precise description of the program **Process** and **Algorithm** are given. Then, a **Strategy Improvement** is detailed, along with the **Updated Algorithm**. Finally, a **Demonstration** of the tool with an existing API will conclude the section.

Regarding the availability of the tool, a *GitHub* repository of *MutGPT* can be found at the following URL: `https://github.com/alixdecr/MutGPT`.

4.1 Overview

In order to fully grasp the overall flow of the utilized strategy before diving into further in-depth explanations, Figure 20 presents an overview of the complete process. Each step of the strategy along with the order of execution is detailed in an orange circle on the figure.

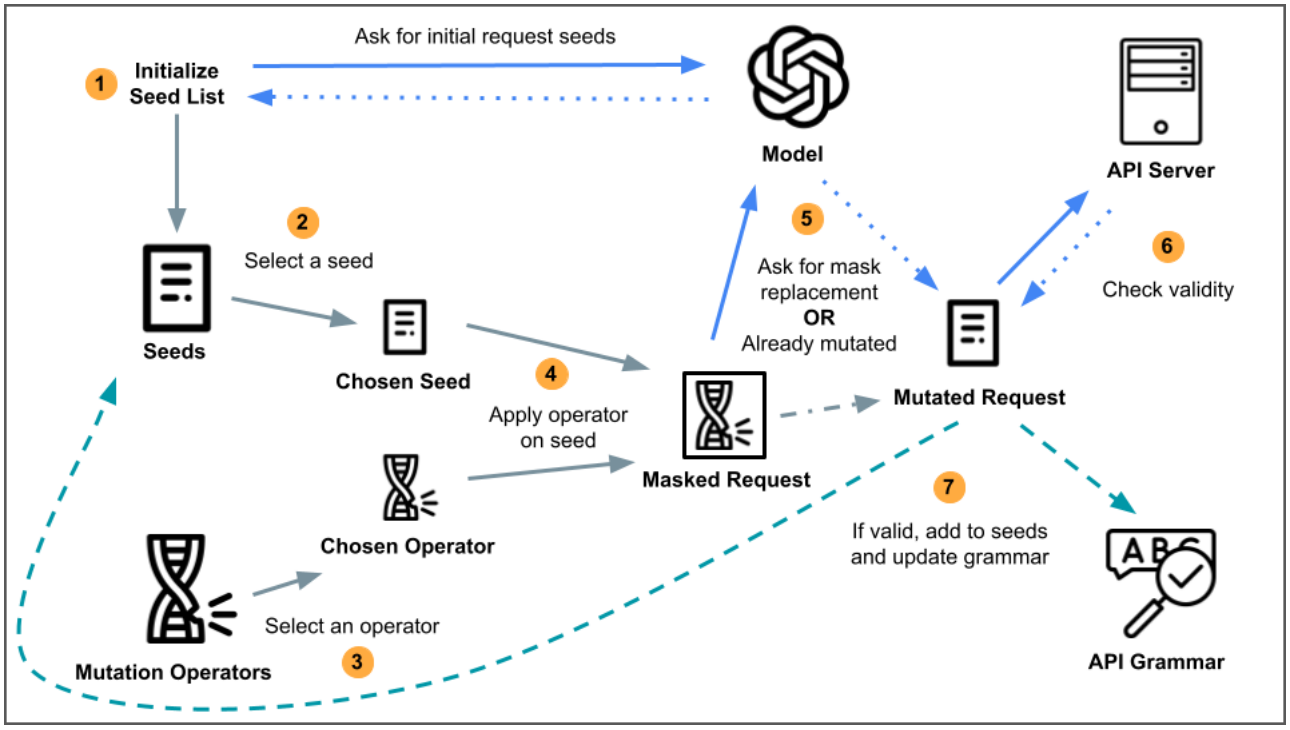


Figure 20: Overview of the Single Request Mutation Strategy process.

At present, a minimal description of the strategy is given in the following paragraph. However, this description only illustrates an overview of *MutGPT*, but does not explain each step in-depth. The complete explanation of the process is given in Section 4.4.

As the purpose of the tool is to generate new HTTP requests based on a given API, step **(1)** begins with the initialization of a seed list. This seed list is generated automatically by prompting the leveraged Large Language Model a first time. In the case of this strategy, the seed list refers to a list of 10 valid HTTP requests. As the seed list has been initialized, the complete iteration of the program can begin. From now on, step **(2)** through **(7)** can be repeated as desired by the program user. Step **(2)** consists in selecting a seed from the previously generated seed list. To complement this selection, step **(3)** involves the selection of a mutation operator. Once both components have been selected, step **(4)** will apply the chosen mutation operator on the selected seed, in order to mask a section of it. The masked request consists in the same request as the chosen seed, however it is slightly modified depending on the chosen mutation operator. This modification consists in the removal of

a section of the request, a replacement of a section with an `<op>` token mask or the addition of such a token. The role of the `<op>` token is to hide a certain section of the request, thus "masking" it. In order to replace this mask with a valid request value, step **(5)** consists in sending the request to the utilized Large Language Model. A certain prompt is also sent with the request, in order for the model to understand that it is supposed to respond with a new valid request, having its mask replaced with a valid value. The response generated by the model is a mutated request, consisting in the initial request slightly modified depending on the mask. However, if the initial request only had a piece of it removed due to a mutation operator, it already consists in a mutated request and does not need to be sent to the model. Moreover, the validity of the mutated request needs to be verified. To do so, step **(6)** consists in sending the mutated request to the utilized API server. As presented in Section 2.1.4, a server response holds information able to indicate the validity of a sent request. Thus, if the request is valid, step **(7)** will add this newly generated request to the initial seed list if it does not already exist in it. This process allows new valid requests to be integrated to the seed list, potentially rendering a more in-depth discovery of the API possible. Finally, step **(7)** also updates the specification[14] of the API, based on the content of the current valid mutated request. This specification is initially empty, however by iteratively complementing it with additional new routes and parameters discovered by mutating requests, the discovery of API specifications is possible and thus automated.

4.2 Mutation Operators

To efficiently mutate requests, various mutation operators are established. These mutation operators allow the modification of various parts of a request such as its routes, parameter names and parameter values. The mutation operators are described in the following paragraphs, each containing a practical example of an initial request, a masked request with the mutation operator and a possible mutation by replacing the mask. The requests given as example may not always be valid, their use being of practical illustration.

addRoute. Mutation operator responsible for adding a new route to a given request. All parameters of the request are kept and are simply shifted to the right after the insertion of the new route.

Example: addRoute

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/<op>/?name=Homo+sapiens&rank=SPECIES`

Possible Mutation: `https://api.gbif.org/v1/species/search/?name=Homo+sapiens&rank=SPECIES`

removeRoute. Mutation operator responsible for removing an existing route from a given request. All parameters of the request are kept and are simply shifted to the left after the removal of the existing route.

Example: removeRoute

Initial Request: `https://api.gbif.org/v1/species/search/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/name=Homo+sapiens&rank=SPECIES`

Possible Mutation: The masked request already consists in the mutated request.

The `removeRoute` mutation operator is only applicable if the given request contains at least 1 base route. Otherwise, the removed route would affect the API base URL, which could presumably result in request errors.

[14] In Figure 20, the specification of the API is referred to the grammar of the API; Both terms are interchangeable.

Example: Invalid use of removeRoute

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/?name=Homo+sapiens&rank=SPECIES`

Possible Mutation: The masked request already consists in the mutated request.

Problem: The mutated request results in a `404 Not Found` HTTP client error message.

modifyRoute. Mutation operator responsible for modifying an existing route of a given request. All other routes and parameters of the request retain their initial structure.

Example: modifyRoute

Initial Request: `https://api.gbif.org/v1/species/search/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/<op>/?name=Homo+sapiens&rank=SPECIES`

Possible Mutation: `https://api.gbif.org/v1/species/match/?name=Homo+sapiens&rank=SPECIES`

The `modifyRoute` mutation operator is only applicable on routes of the request that are not contained in the base URL of the API endpoint.

Example: Invalid use of modifyRoute

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/<op>/species/?name=Homo+sapiens&rank=SPECIES`

Possible Mutation: `https://api.gbif.org/api/species/?name=Homo+sapiens&rank=SPECIES`

Problem: The mutated request results in a `404 Not Found` HTTP client error message.

For the example above, it is possible that when masking the `/v1` route of the request, the model reinserts the same route to replace the mask. This is due to the fact that the model knows that the base URL of the *GBIF Species* API is `https://api.gbif.org/v1/species`, as it was able to generate initial request seeds to this specific endpoint. However, to avoid unwanted errors and ineffectual mutations, the base route of a request is always omitted from modification.

addParameter. Mutation operator responsible for adding a new parameter to a given request. This new parameter consists of a name and a value, of the structure `&name=value`. The added ampersand `"&"` character is used to separate the new parameter from the previous parameter, if there is at least a parameter in the request. Otherwise, the parameter is added as `?name=value` with a question mark `"?"` character, separating the parameter from the rest of the request. Such parameter structure is detailed in Section 2.1.3.

Example: addParameter with a previous parameter

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens`

Masked Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&<op>=<op>`

Possible Mutation: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Example: addParameter without a previous parameter

Initial Request: `https://api.gbif.org/v1/species/`

Masked Request: `https://api.gbif.org/v1/species/?<op>=<op>`

Possible Mutation: `https://api.gbif.org/v1/species/?name=Homo+sapiens`

For the `addParameter` mutation operator, the mask consists of two `<op>` tokens separated by an equal `"="` character in order to retain the structure of a request parameter.

removeParameter. Mutation operator responsible for removing an existing parameter from a given request. Depending on the API in use and mutation preferences, the `removeParameter` mutation operator can only be applied if the given request contains at least a certain predefined amount of parameters. By default, a request needs to contain at least 1 parameter in order for the mutation operator to be utilized.

Example: removeParameter

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/?rank=SPECIES`

Possible Mutation: The masked request already consists in the mutated request.

In the given example, the parameter `name=Homo+sapiens` is removed from the request. As the given request initially contained 2 parameters, the `removeParameter` mutation operator is applicable for a threshold of minimum 1 parameter remaining. The mutated request containing a single parameter left, the ampersand `"&"` character separating both initial parameters is also removed as it is no longer required.

modifyParameter. Mutation operator responsible for modifying an existing parameter of a given request. The modification will retain the initial structure of a request parameter, which is of the form `name=value`.

Example: modifyParameter

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&<op>=<op>`

Possible Mutation: `https://api.gbif.org/v1/species/?name=Homo+sapiens&limit=10`

For the `modifyParameter` mutation operator, the mask consists of two `<op>` tokens separated by an equal `"="` character in order to retain the structure of a request parameter.

modifyParameterName. Mutation operator responsible for modifying an existing parameter name of a given request. The value of the mutated parameter is retained.

Example: modifyParameterName

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&limit=10`

Masked Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&<op>=10`

Possible Mutation: `https://api.gbif.org/v1/species/?name=Homo+sapiens&offset=10`

As shown in the example above, changing a parameter name can allow for the discovery of other parameter names, which contain values of the same type as the initial parameter.

modifyParameterValue. Mutation operator responsible for modifying an existing parameter value of a given request. The name of the mutated parameter is retained.

Example: modifyParameterValue

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&limit=10`

Masked Request: `https://api.gbif.org/v1/species/?name=<op>&limit=10`

Possible Mutation: `https://api.gbif.org/v1/species/?name=Tracheophyta&limit=10`

Comparably to the `modifyParameterName` mutation operator, changing a parameter value enables the discovery of other parameter values, for the same parameter name. The `modifyParameterValue` mutation operator can be very useful, as it can uncover specific parameter values that are not generic. To illustrate this purpose, the parameter `units` of the *OpenWeatherMap* API [63] specifies the desired units of measurement. The values of this parameter are extremely specific, consisting of `standard`, `metric` and `imperial`. If a given request contains a parameter `units=metric`, masking the value `metric` can potentially result in a new parameter `units=imperial`. Thus, a new valid value `imperial` is discovered. However, for generic parameter values such as IDs, usernames or passwords, the value can mostly consist of any arrangement of numbers and characters, which is less interesting.

resetParameters. Mutation operator responsible for resetting all existing parameters of a given request. The `resetParameters` mutation operator can be interpreted as a combination of **(1)** multiple usages of the `removeParameter` operator and **(2)** a single usage of the `addParameter` operator. First, all parameters of a given request are removed. Second, a new parameter mask is added to the request, thus resetting the parameters of the given request.

Example: resetParameters with a single parameter mask

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/?<op>=<op>`

Possible Mutation: `https://api.gbif.org/v1/species/?highertaxon_key=7707728`

For instance, the `resetParameters` mutation operator can discover new parameter combinations, when a given request is "bloated" with a lot of different parameters. Moreover, the mutation operator can reinsert more than 1 new parameter mask in the request, in order to generate a new request containing multiple parameters.

Example: resetParameters with multiple parameter masks

Initial Request: `https://api.gbif.org/v1/species/?name=Homo+sapiens&rank=SPECIES`

Masked Request: `https://api.gbif.org/v1/species/?<op>=<op>&<op>=<op>&<op>=<op>`

Possible Mutation: `https://api.gbif.org/v1/species/?rank=KINGDOM&name=Bacteria&limit=10`

Although the `resetParameters` mutation operator specifies a certain amount of overridden parameters, it is possible for the model to return a request containing more parameters than the prompted amount. This is due to the fact that the model attempts to complete the request with additional parameters based on its knowledge. This completion can be interpreted as a *higher order mutation*, where more than 1 mutation is inserted into the request at a time. However, this supplementation does not always happen. As the model environment is

entirely black-box-based, this matter is only controllable via the input prompt. Nonetheless, such behavior can result in a faster discovery of API parameters.

4.3 Request Validity Verification

4.3.1 Status Code Problem: False Positives

Verifying the validity of mutated request is a crucial task, in order to properly detect the candidate requests that will integrate the seed list and improve the ongoing API inference. As presented in Section 4.1, the validity of requests is verified by sending each mutated request to the corresponding API endpoint, and analyzing the returned status codes. However, returned status codes can in certain cases indicate false positives. To illustrate this purpose, an example is given with the *Helioviewer* API.

To briefly elucidate, the *Helioviewer* Project is a public API allowing access to solar and hemispheric data. As any other API, *Helioviewer* contains various routes to access such data, each containing parameters in order to specify the requested data. During the experimentation process with the Complex Request Prompt Strategy presented in Section 3.2.3, a specificity of the *Helioviewer* API was discovered: *in-page errors*.

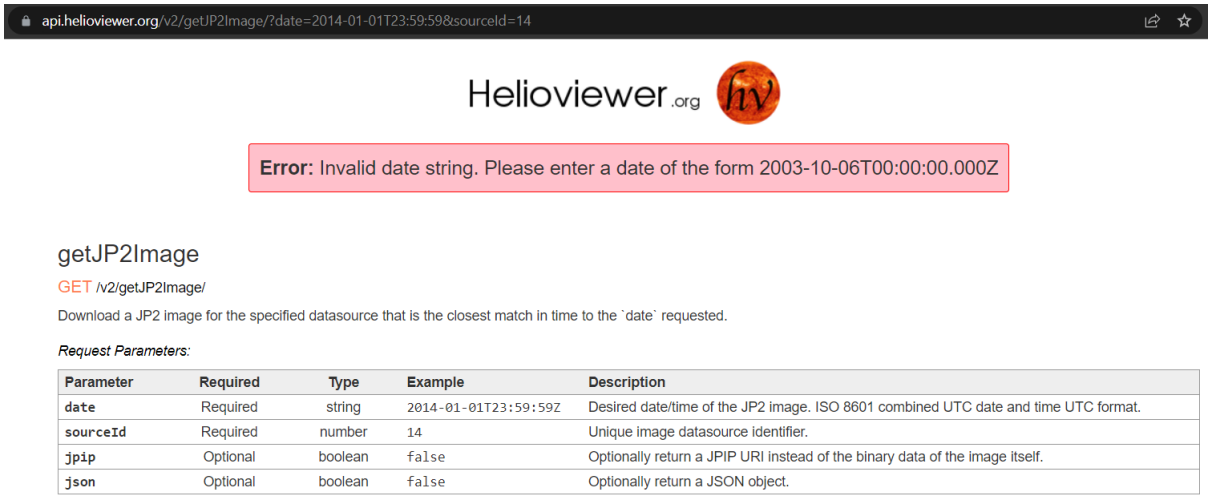


Figure 21: Example of an in-page error of the *Helioviewer* API, when sending a seemingly valid request but with an invalid `date` parameter value.

As displayed in Figure 21, a request is sent to the *Helioviewer* API server, which in response loads a web page on the corresponding website. In the case of the given example, the request is structurally valid enough in order to find the API endpoint. However, the request contains an invalid parameter `date`. As explained on the web page, the `date` parameter requires a specific format with a `"Z"` character at the end of the string, which the request did indeed not contain. For a developer manually inserting a request in a web browser, this response is sufficient to indicate that the sent request is invalid; The displayed in-page error containing a clear and detailed explanation of the issue and how to fix it. Still, when sending a request through a *Python* library and observing the returned status code from the server, this specific example will result in a `200 OK` status code, appearing as if the sent request is valid. In consequence, in-page errors can cause false positives for the validity of mutated requests.

In order to avoid such false positives, further analysis of server responses is required. As the testing environment is entirely black-box-based, the responses returned by the API servers are extremely important, comprising the only available source of information w.r.t. the validity of mutated requests. Even though responses are mostly limited to status codes and returned data, their in-depth analysis allows the discovery of much more interesting elements. The following sections detail the 3 main indicators considered to verify the validity of requests: the **Status Code**, the **Page Title** and the **Response Data**.

4.3.2 Indicator 1: Status Code

As detailed in the previous section, a returned status code consists in the first verification for the validity of a request. Even though status codes can potentially mislead to false positives due to the in-page error issue, a

lot of APIs are built to unmistakably warn that the sent request is invalid. Moreover, a valid request may have caused an in-page error, but this signifies that the API base and routes used in the same request correspond to a valid endpoint of the API. For this first indicator of request validity, requests can be classified into 3 distinct groups depending on the returned status code:

200-299 Status Code Range: Valid Request. A returned status code in this range indicates that the sent request was successfully received and accepted by the API server, regardless of in-page errors. This range group consists of requests that will be added to the seed list and that will be used to infer the specification of the API.

400-499 Status Code Range: Invalid Request due to Client Error. This status code range stipulates that the sent request contained an error caused by the client, due to a bad request syntax or a specified action that could not be fulfilled. This range group comprises requests that are plainly invalid, thus such requests will not be added to the seed list nor update the API specification.

500-599 Status Code Range: Invalid Request due to Server Error. While infrequent for robust APIs, sent requests can potentially cause server errors. This range group contains apparently valid requests that the server failed to fulfil. Requests of this status code group are not retained for the seed list nor the API grammar. However, such requests are listed in a separate file, for a more in-depth manual inspection and for bug reproduction.

In consequence, the 3 status code groups allow for a preliminary classification and elimination of generated requests.

4.3.3 Indicator 2: Page Title

To avoid false positives of request status codes, analyzing the content data returned by the server can lead to an increased confidence in the validity of requests. An interesting aspect to consider is that if a request generated an in-page error, the response content from the server will correspond to a HTML representation of a web page for this in-page error. Thus, by parsing the HTML document, it is plausible to detect error messages. A straightforward way of doing so is to simply analyze the content of the HTML page title. This title is always found in a `<title>` HTML tag; By reducing the search space to this field, checking for an "error" string can result in an adequate indicator of an in-page error. Listing 2 illustrates an example of a HTML web page returned by the *Helioviewer* server when sending a request. The request being invalid, the HTML document represents an in-page error describing the issue contained in the given request. The `<title>` tag displays the message `"Helioviewer.org API - Error"`, which can be classified as an in-page error in the program by parsing the title.

While this simple indicator for the validity of a request can prove to be performing, it should be treated with caution. Indeed, it is possible that page titles do not accurately reflect the validity of a request. For this reason, when *MutGPT* detects a potential in-page error for a valid request according to the status code, it will not render the request invalid but will display a warning message during the mutation process.

4.3.4 Indicator 3: Response Data

While HTML responses often represent a web page, it is not always the case, especially for APIs. Frequently, when sending a GET HTTP request to an API endpoint, the server will only respond with the required data. In such cases, a `<title>` tag is nonexistent, rendering the page title parsing unsuccessful. Listing 3 and Listing 4 display examples of data returned by the *OpenWeatherMap* API and *GBIF Species* API, respectively, when sending valid GET HTTP requests.

Nonetheless, this returned data can consist in a supplemental indicator of the validity of a request. First of all, if a request resulted in a status code in the 200-299 range, it hitherto signifies that the API server has accepted the request. Second of all, if response content returned by the API server is data contained in a JSON[15] object, it most likely corresponds to the requested data.

By combining status codes, page titles and response data, it becomes possible to have a respectable criterion of the validity of a generated request. Such procedure of analyzing the validity of a request is contained in the implemented program, and is executed during the request validity verification step displayed in Figure 20. For

[15]While APIs often return a JSON object for data representation, other data formats can also be used.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- DATE: 2023-05-20 14:05:11 URL: http://api.helioviewer.org/v2/stats/getNumFilters
    /?start_time=image&appid= -->
    <title>Helioviewer.org API - Error</title>
    ...
</head>
<body>
    ...
    <b>Error:</b> Invalid action specified.<br />Consult the <a href=
    "https://api.helioviewer.org/docs/">API Documentation</a> for a list of valid actions.
    ...
</body>
</html>
```

Listing 2: Example of a HTML code snippet representing a web page returned by the *Helioviewer* API server after sending a request. The `<title>` tag indicates an in-page error. An error message is also found in the HTML body.

```
{"coord":
    {"lon":24.0833,
     "lat":57},
 "weather":
    [{"id":800,
      "main":"Clear",
      "description":"clear sky",
      "icon":"01d"}],
 "base":"stations",
 ...
 "name":"Rīga",
 "cod":200}
```

Listing 3: Example of JSON data returned by the *OpenWeatherMap* API server following a valid GET HTTP request.

```
{"key":7,
 "nubKey":7,
 "nameKey":97526282,
 "taxonID":"gbif:7",
 "sourceTaxonKey":170809337,
 "kingdom":"Protozoa",
 ...
 "lastInterpreted":"2022-11-23T06:42:12.719+00:00",
 "issues":[],
 "synonym":false}
```

Listing 4: Example of JSON data returned by the *GBIF Species* API server following a valid GET HTTP request.

each generated request, the status code, the page title and the response data are displayed by *MutGPT* in the console for the user to easily follow the validity verification process.

4.4 Process

As preliminary understandings of the program have been detailed in the previous sections, the complete process of *MutGPT* can now be presented. For the purpose of rendering the strategy fully intelligible, each step of the process is detailed in the following paragraphs. Along with each presented step, an example is given with an existing API. The strategy utilized by *MutGPT* is labeled as the **Single Request Mutation Strategy (SRMS)**.

Preliminary Step: Program Parameters. This preliminary course of action is required in order for the program to understand which API is used. The process being previously described as fully automated, the only user inputs that are required consist in:

- **API Name**: In order to generate requests, the program needs to know the name of the relevant API.
- **API Key**: If the API requires a key to be specified as parameter for authentication, it needs to be given as input to the program, which will make sure to always include it when generating requests.
- **API Base**: This parameter specifies the base route of the API, for the purpose of sending requests to the adequate endpoint.
- **Number of Mutations**: This amount corresponds to the number of required mutations, representing the number of iterations and thus the number of attempted generated requests.
- ***OpenAI* Key**: To leverage *OpenAI*'s Large Language Models, a secondary key is required as input. This key can be generated and found in the user's *OpenAI* account, as described in Section 3.1.2.

With these inputs given, the program is entirely set up to generate requests and infer the API specification. From now on, the entire process is self-executing and no longer requires user input. Moreover, the model leveraged by *MutGPT* is the *text-davinci-003* model.

Example: Program Parameters

The following values are given as input to the program:

- **API Name**: *OpenWeatherMap* API
- **API Key**: f1484466d24084c53f150515f95xxxxx
- **API Base**: `https://api.openweathermap.org/data/2.5`
- **Number of Mutations**: 50
- ***OpenAI* Key**: sk-EgAMaRK1jxDC2wxXOGt2T3BlbkFJ9v35HLHTTS2XeZpxxxxx

The 5 last characters of the keys are replaced with "xxxxx" for privacy purposes. In order to render future examples comprehensible, the API key will be directly replaced with "xxxxx" in the following instances.

Step 1: Seed Initialization. Considering that no initial request seeds are required in the user input step, an initial operation consists in generating such seeds for the purpose of mutating requests. Section 3.2 presented 2 strategies in order for Large Language Models to generate requests: the BRPS and the CRPS. While such strategies proved to comprise issues when generating numerous requests, they can still be of use for the purpose of generating a couple of initial request seeds. Thus, the first step of *MutGPT* consists in leveraging the Complex Request Prompt Strategy described in Section 3.2.3 to generate a predefined amount of 10 request seeds.

However, it is possible for the model to return invalid request seeds, thus rendering the future mutation process inoperative. For this reason, the validity of the generated seeds is verified. If not all generated requests are valid, the seed list will only contain the requests that are valid, as long as there is at least 1 valid request. If no request proved to be valid, *MutGPT* will attempt to re-prompt the model for new valid requests. If this process also results in a complete list of invalid requests, the program user will be asked to provide request

seeds manually. In spite of such potential issue, the program has to date never failed to automatically generate request seeds, with over 15 different APIs tested. Whilst the *text-davinci-003* model is seemingly well-trained for popular APIs such as the *Spotify* API [77], it proved to also generate valid request seeds for the *GBIF Species* API [73], the *Random User Generator* API [66], the *Chuck Norris Jokes* API [41], the *Helioviewer* API [47] and other tested APIs described in Section 5.

Example: Seed Initialization

The program prompts the leveraged model with the following task:

Model Prompt:

"Generate 3 examples of complex GET HTTP requests that can be made to the `OpenWeatherMap API` web service. The example requests need to be in a Python list of the following structure: `["request 1", "request 2", ...]`. The requests should all be valid, have various routes and different parameters. The requests cannot be too similar. Do not forget to include the base route of the API in each request. The API key to be used in requests is the following: `xxxxx`."

The model then responds with the following data:

Response:

```
["https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx",
 "https://api.openweathermap.org/data/2.5/weather?lat=44.34&lon=10.99&appid=xxxxx",
 "https://api.openweathermap.org/data/2.5/rain?q=Moscow&appid=xxxxx"]
```

Next, the program analyzes the validity of each request:

Validity:

- Request 1: `200 OK` status code, the request is valid.
- Request 2: `200 OK` status code, the request is valid.
- Request 3: `404 Not Found` status code, the request is invalid as the `rain` route does not exist in the API.

Finally, the valid requests are added to the seed list:

Seed List:

```
["https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx",
 "https://api.openweathermap.org/data/2.5/weather?lat=44.34&lon=10.99&appid=xxxxx"]
```

For clarity purposes, only 3 requests are specified in the model prompt of this example, in contrary to a usual amount of 10 requests.

Step 2: Seed Selection. As initial request seeds have been generated, the following step consists in selecting a seed at random. The chosen seed will be the candidate request for the current mutation in the iteration process.

Example: Seed Selection

A request is chosen at random from the seed list:

Chosen Seed: `https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx`

Step 3: Mutation Operator Selection. As a seed has been selected, a mutation operator is now selected for the seed. The selection is not completely random, as certain mutation operators are ruled out by the program depending on the given seed. For instance, if the chosen request seed does not contain a single parameter, the mutation operators responsible for removing and modifying parameters are removed from the selection.

Example: Mutation Operator Selection

Based on the selected seed, certain mutation operators are ruled out of the selection:

Omitted Mutation Operators:

- `removeRoute`, as `weather` is a base route of the API.
- `removeParameter`, as `q` is the only parameter besides the mandatory API key parameter `appid`.

A mutation operator is now randomly selected from the remaining mutation operators:

Remaining Mutation Operators: `addRoute, modifyRoute, addParameter, modifyParameter, modifyParameterName, modifyParameterValue, resetParameters`

Chosen Mutation Operator: `modifyParameterValue`

Step 4: Request Masking. The mutation operator is now applied on the given request. Depending on the chosen mutation operator, there exists 2 possible courses of action:

- **Mutation Operator of Type Remove**: If the mutation operator consists in removing a route or a parameter, the masked request will consist in the same request diminished of a route or a parameter.
- **Other Mutation Operator**: However, if the mutation operator specifies that an element of the request needs to be modified or supplemented, an `<op>` token mask is inserted accordingly in the given request. The masked request will consist in the request with an added or supplemented `<op>` token mask.

Example: Request Masking

The chosen request seed is masked based on the `modifyParameterValue` mutation operator, which consists in modifying an existing parameter value of the request. Since `q` is the only available parameter besides the mandatory API key parameter `appid`, its value is masked:

Masked Request: `https://api.openweathermap.org/data/2.5/weather?q=<op>&appid=xxxxx`

Step 5: Request Mutation. Depending on the masked request produced in the previous step, 2 different flows are attainable:

- **Masked Request does not Contain an `<op>` Token**: If the previously masked request was constructed with a mutation operator of type remove, it does not contain an `<op>` token. Hence, the masked request already corresponds to the mutated request and does not need to be sent to the model.
- **Masked Request Contains an `<op>` Token**: If the masked request contains an `<op>` token mask, this signifies that the placeholder token necessitates a replacement. Therefore, the current step consists in prompting the leveraged Large Language Model with the following task:

Prompt: Single Request Mutation Strategy

"Replace the `<op>` token mask in the following request: `MASKED_REQUEST`. Give me the complete new request which needs to be different from the following previous request: `INITIAL_REQUEST`. Only replace the `<op>` tokens in the request and nothing else. When the `<op>` token is a route or a parameter, try to find new valid routes and new parameters that have not been used yet. Do not invent route and parameter names, only answer with valid routes and parameters that exist in the API documentation."

If an API key is required, add: "Always include the following API key as parameter: `API_KEY`."

Parameters:

- `MASKED_REQUEST`: Request containing the mask.
- `INITIAL_REQUEST`: Initial request, without the mask.
- `API_KEY`: Key to be specified in API requests, if required.

Leveraged Model: *text-davinci-003*

Example: Request Mutation

The program prompts the leveraged model with the following task:

Model Prompt:

"Replace the `<op>` token mask in the following request:

`https://api.openweathermap.org/data/2.5/weather?q=<op>&appid=xxxxx`.

Give me the complete new request which needs to be different from the following previous request:

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx`.

Only replace the `<op>` tokens in the request and nothing else. When the `<op>` token is a route or a parameter, try to find new valid routes and new parameters that have not been used yet. Do not invent route and parameter names, only answer with valid routes and parameters that exist in the API documentation."

The model then responds with the following mutated request, containing a modified parameter value:

Mutated Request:

`https://api.openweathermap.org/data/2.5/weather?q=New+York,us&appid=xxxxx`

Example: Masked Request is already the Mutated Request

However, if the chosen mutation operator was `removeRoute`, which consists in removing an existing route from the request, the masked request is already the mutated request. For illustration purposes, the `weather` route of the chosen request seed is removed:

Masked Request: `https://api.openweathermap.org/data/2.5?q=London,uk&appid=xxxxx`

The masked request does not need any replacement, as it does not contain an `<op>` token mask. Thus, it is not sent to the leveraged Large Language Model and the mutated request is:

Mutated Request: `https://api.openweathermap.org/data/2.5?q=London,uk&appid=xxxxx`

Step 6: Request Validity Verification. In order to verify if the newly generated mutated request is valid, it is sent to the corresponding API endpoint. If the API server replies with a status code in the 200 range, the request is valid. Otherwise, the request is invalid. The program will also check if the server HTTP response contains a page title for potential in-page errors, and if the returned data is in a JSON format.

Example: Request Validity Verification

The mutated request is sent to the *OpenWeatherMap* API server endpoint as a GET HTTP request. The server then replies with a HTTP response:

Server Response: `200 OK` status code with data


```
{"coord":
    {"lon":-74.006,
     "lat":40.7143},
 "weather":
    [{"id":800,
      "main":"Clear",
      "description":"clear sky",
      "icon":"01d"}],
 "base":"stations",
 ...}
```


Step 7: New Seed and Specification Update. If the mutated request succeeded the previous request validity verification, it is first added to the existing request seeds, if it does not yet exist in the list.

Example: Updating the Seed List

As the mutated request does not yet exist in the seed list, it is added to it:

Updated Seed List:


```
["https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx",
 "https://api.openweathermap.org/data/2.5/weather?lat=44.34&lon=10.99&appid=xxxxx",
 "https://api.openweathermap.org/data/2.5/weather?q=New+York,us&appid=xxxxx"]
```


Moreover, the valid mutated request is decomposed by the program for the purpose of analyzing its structure and further inferring the API specification. The API specification is initially empty, however each generated request can add new undiscovered elements to it such as:

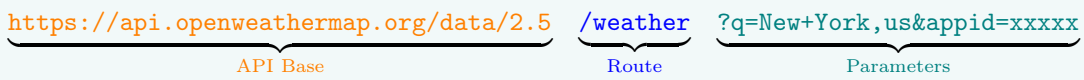
- Routes
- Parameter names
- Parameter values

In order to retain inference progress throughout multiple program executions, the API specification is stored in an external file. The file is always named after the given API name, followed by `_grammar.txt`. The seed list is also contained in a file, named after the given API name and followed by `_seeds.txt`. Moreover, the specification is contained in a JSON object; The base of the API is located at the base of the object, and each route is contained in the base. The same structure also applies to parameters, which are contained in their relevant routes. All different parameter values are stored in a list. Listing 5 displays a representation of the structure of an API specification JSON object.

Example: Updating the API Specification

First, the valid mutated request is decomposed into sections:

Request Decomposition:



The API base, the routes and the parameters of the request are separated. The parameters are also decomposed into smaller sections:

Parameter Decomposition:

By decomposing the request into smaller sections, a part of the API specification can be inferred by the program:

API Specification:

```
{
    "https://api.openweathermap.org/data/2.5": {
        "/weather": {
            "q": ["New+York,us"],
            "appid": ["xxxxx"]
        }
    }
}
```

As the API specification was empty beforehand, all sections of the request have been added. As parameters can have multiple possible values, the value of each parameter has been added into a list, in order to potentially insert other future parameter values into it. Indeed, if during a next iteration step a new valid mutated request is found:

New Request: `https://api.openweathermap.org/data/2.5/weather?q=Berlin&appid=xxxxx`

It will also be decomposed by the program. When analyzing the existing API specification, it is apparent that all routes and parameter names of the request already exist. However, for the parameter `q`, a new value `Berlin` can be added to the specification:

Updated API Specification:

```
{
    "https://api.openweathermap.org/data/2.5": {
        "/weather": {
            "q": [
                "New+York,us",
                "Berlin"
                ],
            "appid": ["xxxxx"]
        }
    }
}
```

```
{
    API_BASE: {
        ROUTE_1: {
            PARAMETER_1: [
                VALUE_1,
                VALUE_2,
                ...
                ],
            PARAMETER_2: [...],
            ...
        }
        ROUTE_2: {...}
        ...
    }
}
```

Listing 5: Structure of an API specification JSON object.

4.5 Algorithm

The algorithm of *MutGPT* is presented as pseudocode in Listing 6. The algorithm represents an overview of the actual program code, which does not accurately reflect *MutGPT*. For instance, strings contained in the `prompt_llm` method do not reflect the actual precision of the prompted tasks. Nonetheless, the utilized prompts have been thoroughly described in Section 3.2.3 for the request seed generation, and in Section 4.4 for the request mutations. Moreover, the employed method names are used for illustration purposes and do not reflect the actual methods implemented in the program.

4.6 Strategy Improvement: Multiple Mutations with a Single Request

While the Single Request Mutation Strategy proved to be particularly effective at generating new requests and inferring API specifications, an issue persisted: optimization.

In fact, each iteration in the mutation process requires an API request to be sent to the *OpenAI* server. For a couple of mutations this issue is trivial, however inferring the specification of large APIs requires a lot of mutations with the current strategy. As explained in Section 3.1.3, sending requests to the *OpenAI* API server costs a certain amount of credit, but can be also costly in terms of response time. For this reason, an improved and upgraded version of *MutGPT* was implemented, relatively solving the optimization issue. Figure 22 presents an overview of the new and improved request mutation strategy. This improved strategy is labeled as the **Multiple Request Mutation Strategy (MRMS)**. In order to fully understand the changes in the process, each new step is detailed in the following paragraphs. Similarly to the initial request mutation strategy, a practical example with an existing API is given in parallel for each step.

Preliminary Step: Required User Input. The preliminary course of action required by the program is almost identical to the one described in the Single Request Mutation Strategy. To recall, the program needs to be given as input the name of the API, the key of the API if necessary, the API base, the number of required mutations and the *OpenAI* key. However, the improved program does not require the API base as input anymore, as *MutGPT* will be responsible for automatically discovering the base of the input API without the user having to do so.

Step 1: API Information Initialization. By further analyzing the capabilities of the leveraged *text-davinci-003* model, additional automation is possible. When launching the program, *MutGPT* will automatically find relevant URLs of the utilized API. In order to do so, the following tasks are prompted to the model:

```
# get program parameters from the user
api_input = get_user_input()
nb_mutations = get_user_input()
openai_key = get_user_input()

# initialize the seed list
request_seeds = prompt_llm("Generate requests for", api_input, openai_key)
seed_list = []
for seed in request_seeds
    # send seed request to API server
    response = send(seed)
    if response is valid
        seed_list += seed

# iterate for request mutations
for mutation in nb_mutations
    chosen_seed = random(seed_list)
    filter_mutation_operators(chosen_seed)
    chosen_operator = random(mutation_operator_list)

    masked_request = mask(chosen_seed, chosen_operator)

    if masked_request is not mutated
        mutated_request = prompt_llm("Replace the mask of", masked_request, chosen_seed,
        openai_key)

    else
        mutated_request = masked_request

    # send mutated request to API server
    response = send(mutated_request)

    if response is valid
        if title in response has "error"
            warn_user("Valid request contains a potential in-page error")

        if mutated_request not in seed_list
            seed_list += mutated_request
            infer_api_specification(mutated_request)
```

Listing 6: Algorithm of *MutGPT*, represented in pseudocode.

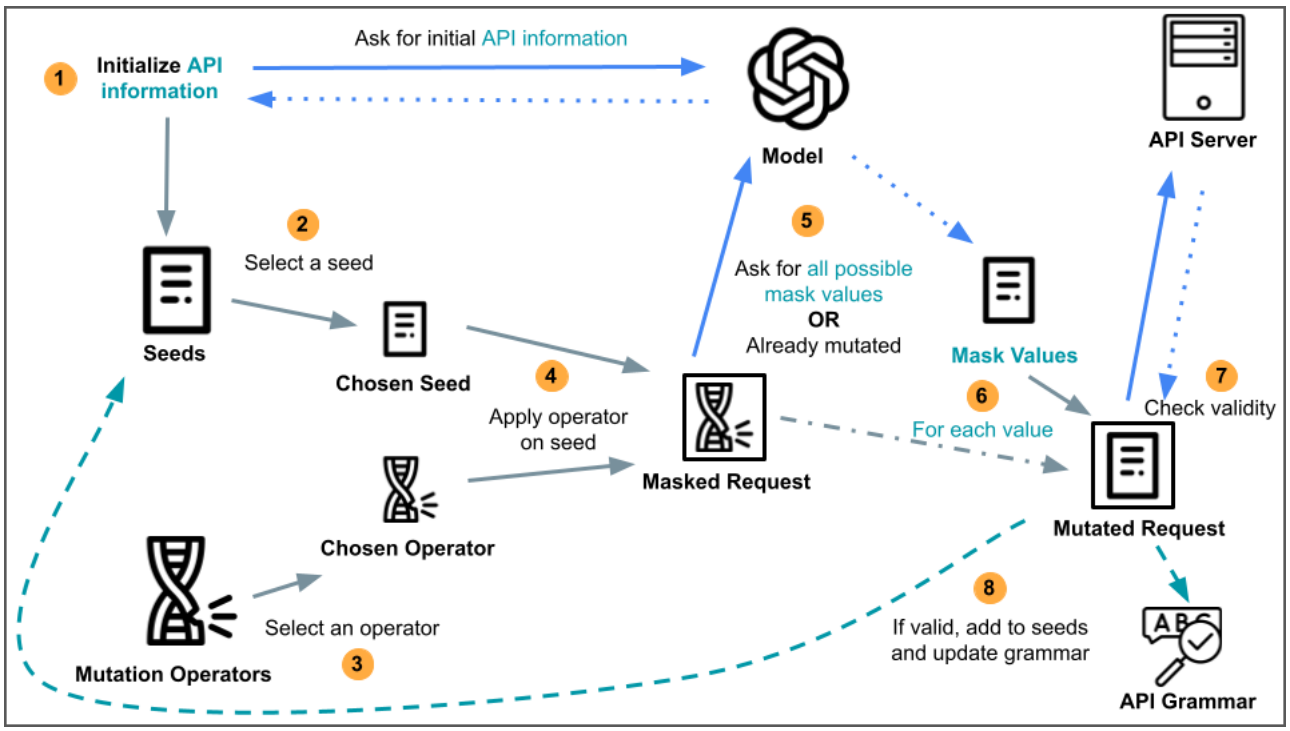


Figure 22: Overview of the Multiple Request Mutation Strategy process. Changes are colored in teal.

Prompt: API Information Initialization

Base URL: "Give me the base URL for requests to the `API_NAME`. Do not forget to include the version of the API in the URL if it exists."

Documentation URL: "Give me the URL of the documentation for API requests to the `API_NAME`."

Initial Seed: "Give me an example of a valid GET HTTP request containing various parameters that can be made to the `API_NAME`."

Parameter:

- `API_NAME`: Name of the API.

Leveraged Model: *text-davinci-003*

As displayed above, the utilized *OpenAI* model is capable of automating the process even further by returning the API base URL, the API documentation URL and an initial request seed. As each prompted element consists in a URL, the program is able to easily parse these elements in the model response with a basic regular expression beginning with `https://`. The base URL is used to generate requests, while the documentation is displayed to the *MutGPT* user presuming it would serve as a practical source of information. As for the initial seed, the present strategy proved that a single request seed is sufficient in order to efficiently generate mutated requests, in contrary to the 10 required seeds from the previous strategy. Since a single seed is generated, its validity also needs to be verified by the program. If this seed is not valid, the program will attempt to retry the process of prompting the model for a new seed and then verifying its validity. However, if the process results in another invalid seed, the program will use the API base URL as a default seed. In fact, the base URL of an API can be considered as a seed, as mutation operators such as `addRoute` and `addParameter` are able to expand from the base and find new routes and parameters respectively. This decrease in the required initial seeds also contributes to a reduced model response time. In addition, the found URLs are stored in an external file for future use. The file is always named after the given name of the API, followed by `_base.txt`.

Example: API Information Initialization

For the *OpenWeatherMap* API, the model is able to return the following valid URLs:

Base URL: `https://api.openweathermap.org/data/2.5`

Documentation URL: `https://openweathermap.org/api`

Initial Seed: `https://api.openweathermap.org/data/2.5/weather/?q=Paris&cnt=7&appid=xxxxx`

The base URL of the API also contributes to another operation. When the model generates such URL, *MutGPT* will analyze the number of "base routes" contained in it. This analysis allows the program to distinguish additional routes in mutated requests, that do not consist in routes contained in the API base. Moreover, such behavior allows the program to eliminate certain mutation operators during the mutation operator selection step. The following example illustrates this action.

Example: API Base Routes Detection

Initially, the program will store the number of base routes contained in the API base URL:

Base Routes: `https://api.openweathermap.org/data/2.5` contains 2 base routes (`/data` and `/2.5`).

Further in the program iteration process, the following request seed is chosen:

Chosen Request Seed: `https://api.openweathermap.org/data/2.5?q=London,uk&appid=xxxxx`

By parsing the request seed, the program detects that it contains 2 routes. However, as the number of request routes is less than or equal to the number of API base routes, 2 mutation operators are ruled out:

Omitted Mutation Operators: `removeRoute` and `modifyRoute`

As both of these mutation operators would modify the base URL of the API, which would result in an invalid request by all means.

Step 2, Step 3 and Step 4. The seed selection, mutation operator selection and request masking processes do not change from the initial request mutation strategy. The mutation operator process has slightly been modified by incorporating the number of API base routes, as described in the previous step. The flow of the program remains the same, where a seed and a mutation operator are selected, and then the chosen seed is masked by applying the mutation operator onto it.

Step 5: Prompting the Model for All Mask Values. The core improvement of the Multiple Request Mutation Strategy lies in the current step. While executing mutations with the initial Single Request Mutation Strategy, the following question was asked: What if instead of prompting the model for a single value to replace the mask of a given request, all possible replacement values could directly be returned? This procedure would allow for a sizeable reduction of sent *OpenAI* requests, thus partially solving the previously described issues. Moreover, requesting all possible mask values directly would accelerate the discovery of new mutated requests, thus improving the inference rate of API specifications.

In order to do so, the following new task is prompted to the Large Language Model:

Prompt: Multiple Request Mutation Strategy

"Give me a Python list only containing all possible and valid values that can replace the <op> token mask in the following request: MASKED_REQUEST. Here is an explanation: OPERATOR_DESCRIPTION. Make sure that each value is valid for the given API, do not invent any routes and parameters, only use real ones that exist in the API. Make sure that your answer is a Python list of the following structure: ["value1", "value2", ...]."

Parameters:

- MASKED_REQUEST: Request containing the mask.
- OPERATOR_DESCRIPTION: Text describing the mutation operator applied onto the request.

Leveraged Model: *text-davinci-003*

As shown above, the prompt drastically changes from the prompt of the Single Request Mutation Strategy. In this improved version, a mutated request based on the mask is not asked, but rather a *Python* list containing all possible values that can replace the mask. In order for the model to return adequate values, a short description of the mutation operator applied on the given request is given. This description is represented by OPERATOR_DESCRIPTION in the prompt. For the parsing process of the program, the prompt insists on the fact that the model should return the found values in a *Python* list data structure.

Example: Prompting the Model for All Mask Values

For this example, a request seed has been masked with the addParameter mutation operator, consisting in adding a new parameter to a request. The program prompts the leveraged model with the following task, also containing a short description of the addParameter mutation operator:

Model Prompt:

"Give me a Python list only containing all possible and valid values that can replace the <op> token mask in the following request:

https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&<op>=<op>.

Here is an explanation: The <op>=<op> token is masking a parameter name and value. Give me all possible parameter names and values that can replace the <op>=<op> token placeholder. The given answers need to be strings of the structure "parameter=value", with example values for each parameter. Make sure that each value is valid for the given API, do not invent any routes and parameters, only use real ones that exist in the API. Make sure that your answer is a Python list of the following structure: ["value1", "value2", ...]."

The model then returns the following 4 new parameters, contained in a *Python list*:

Mask Values:

```
["units=metric", "lang=de", "mode=json", "cnt=7"]
```

As displayed in the example above, the improved strategy allows the model to respond with 4 new potentially valid parameter names and values, in contrary to a single mutated request with the initial strategy. As no specific amount of mask replacement values is specified, the quantity of returned values in the response list can vary. However, the required amount of *OpenAI* requests for a single masked request is reduced by the amount of values returned in the list, as each new seed would have previously required separate *OpenAI* requests in order to discover the parameters. Thus, this solution greatly improves the optimization issue.

It is important to be mindful of the fact that similarly to the Single Request Mutation Strategy, if the

mutation operator does not add an <op> token mask to the request seed, the masked request does not need to be sent to the model. Indeed, such mutation operators consist in removing routes or parameters, which do not require new values to be discovered.

Step 6: Generating All Mutated Requests. As mask value replacements have been returned by the model in a list, the mutated requests based on the chosen seed and mask still need to be generated. In order to reduce the amount of tokens contained in the model responses - and thus reducing the cost of leveraging the model, the previous step only demands the replacement values from the model. The mutated requests can then be generated by the program itself, as mutating the masked request with given values is a trivial task. The program will simply parse the masked request in order to find the <op> token mask, and then generate new variations of requests by replacing the mask with the returned values.

Example: Generating All Mutated Requests

With the masked request and the mask values returned by the model, the program is capable of generating the mutated requests:

Masked Request:

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&<op>=<op>`

Mask Values:

```
["units=metric", "lang=de", "mode=json", "cnt=7"]
```

Mutated Requests:

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&units=metric`

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&lang=de`

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&mode=json`

`https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=xxxxx&cnt=7`

Step 7: Request Validity Verification of All Mutated Requests. In order to verify the validity of the mutated requests, each mutated request is sent to the corresponding API endpoint. The process is identical to the request validity verification step of the Single Request Mutation Strategy, but with multiple mutated requests to check. For each request, the status code, page title and response data is analyzed and displayed.

Example: Request Validity Verification for All Mutated Requests

For each mutated request generated in the previous example, the program will analyze their validity and returned content:

Validity:

- Request 1: `200 OK` status code with JSON data returned, the request is valid.
- Request 2: `200 OK` status code with JSON data returned, the request is valid.
- Request 3: `200 OK` status code with JSON data returned, the request is valid.
- Request 4: `200 OK` status code with JSON data returned, the request is valid.

Step 8: New Seeds and Specification Updates. The last step consists in adding the valid mutated requests to the seed list, for requests that do not exist in the list yet. As the process is identical to the initial

Request Mutation Strategy and simply consists in adding requests to a *Python* list, no example is given.

Moreover, all valid mutated requests are analyzed by the program, in order to further infer the existing API specification. As mutated requests originating from the same masked request have the same mask element modified, the inference for "batches" of mutated requests will vary depending on the chosen mutation operator. For instance, certain sets of mutated requests will infer new values for a parameter, while other sets will infer new routes of the API.

Example: Updating the API Specification with All Mutated Requests

For each valid mutated request in the previous example, the following API specification can be inferred, containing various new parameters:

Updated API Specification:

```
{
    "https://api.openweathermap.org/data/2.5": {
        "/weather": {
            "q": ["London,uk"],
            "appid": ["xxxxx"],
            "units": ["metric"],
            "lang": ["de"],
            "mode": ["json"],
            "cnt": [7]
        }
    }
}
```

4.7 Updated Algorithm

As the updated version of the request mutation strategy has been detailed, Listing 7 presents an updated version of the program algorithm in pseudocode. Comparably to the initial algorithm, the pseudocode does not reflect the actual content of the program, and is simplified for comprehension purposes.

4.8 Demonstration

Following the complete exposition of *MutGPT*, its algorithm and improvements, a demonstration of the tool is now illustrated. This demonstration will exemplify the complete process of the tool, from the perspective of a user wanting to infer the specification of an API. The *OpenWeatherMap* API will be used as example.

Launching the Tool and Specifying Parameters. By launching the tool in a command prompt, *MutGPT* will ask for the prerequisite parameters.

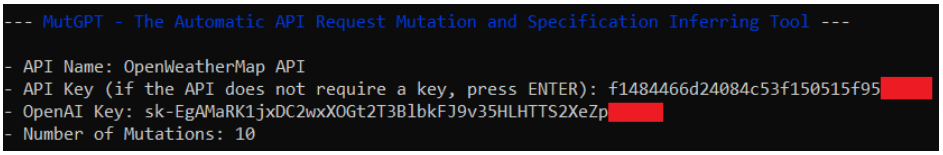


Figure 23: Example of the parameter interface displayed in a command prompt when launching *MutGPT*. The API key and the *OpenAI* key have their 5 last characters hidden for privacy purposes.

As shown in Figure 23, the name of the *OpenWeatherMap* API is given, along with the API key, the *OpenAI* key and the number of required mutations. For demonstration purposes, only 10 mutations are specified.

Finding Initial API Information. As presented in Section 4.6, *MutGPT* is able to automatically find the API base URL, the API documentation URL and an example of a request seed.

```
# get program parameters from the user
api_input = get_user_input()
nb_mutations = get_user_input()
openai_key = get_user_input()

# find API URLs
api_base_url = prompt_llm("Give me the base URL of", api_input, openai_key)
nb_base_routes = len_routes(api_base_url)
api_doc_url = prompt_llm("Give me the documentation URL of", api_input, openai_key)
request_seed = prompt_llm("Generate a request for", api_input, openai_key)

# initialize the seed list
seed_list = []
# send seed request to API server
response = send(request_seed)
if response is valid
    seed_list += request_seed
else
    retry

if seed_list is empty
    seed_list += api_base_url

# iterate for request mutations
for mutation in nb_mutations
    chosen_seed = random(seed_list)
    filter_mutation_operators(chosen_seed, nb_base_routes)
    chosen_operator = random(mutation_operator_list)

    masked_request = mask(chosen_seed, chosen_operator)

    # generate all mutated requests based on replacement values
    mutated_requests = []

    if masked_request is not mutated
        mask_values = prompt_llm("Give me all values that can replace", masked_request,
        mutation_operator_description, openai_key)

        for value in mask_values
            mutated_requests += mutate(masked_request, value)

    else
        mutated_requests += masked_request

    for mutated_request in mutated_requests
        # send mutated request to API server
        response = send(mutated_request)

        if response is valid
            if title in response has "error"
                warn_user("Valid request contains a potential in-page error")

            if mutated_request not in seed_list
                seed_list += mutated_request
                infer_api_specification(mutated_request)
```

Listing 7: Updated algorithm of *MutGPT*, represented in pseudocode.

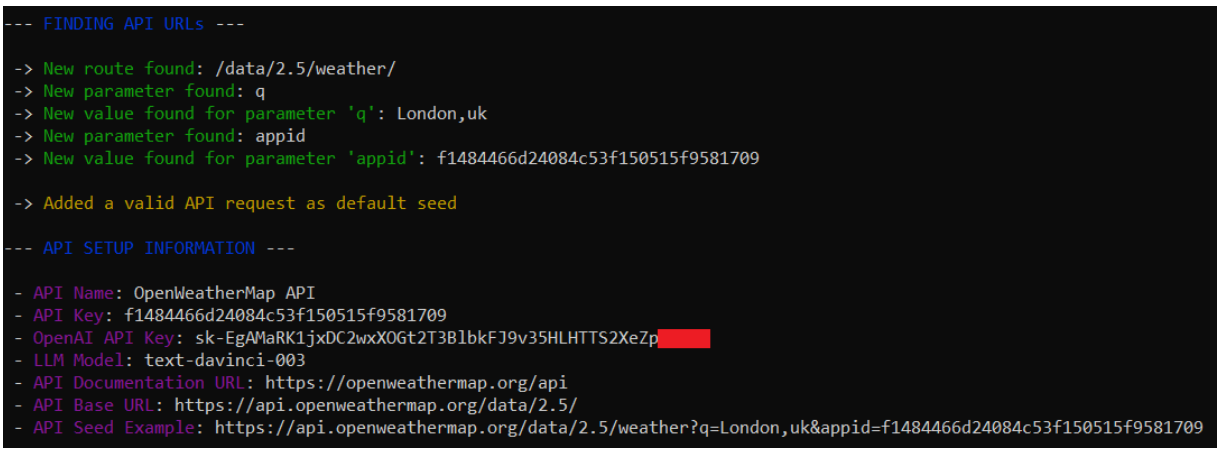


Figure 24: Example of the interface displayed by *MutGPT* when automatically finding the API base URL, the API documentation URL and a request seed.

Figure 24 displays the results found by *MutGPT* for the *OpenWeatherMap* API. First, a title `FINDING API URLs` is shown, describing that it is currently prompting the Large Language Model in use to automatically discover API information. Below the title, text is displayed in green, signifying that a valid request seed has been found and that its elements are being used to infer the specification of the API. A yellow message is also displayed, informing the user that a valid request seed has been added to the seed list. Another title is displayed beneath, `API SETUP INFORMATION`, revealing the complete setup parameters and information for the API in use. The last lines display the 3 URLs that have been automatically found by the program.

First Request Mutation. As the program is entirely automated following the insertion of the required *MutGPT* parameters, the request mutation process will begin. In order for the user to keep track of what is happening, *MutGPT* will display information regarding the current mutation.

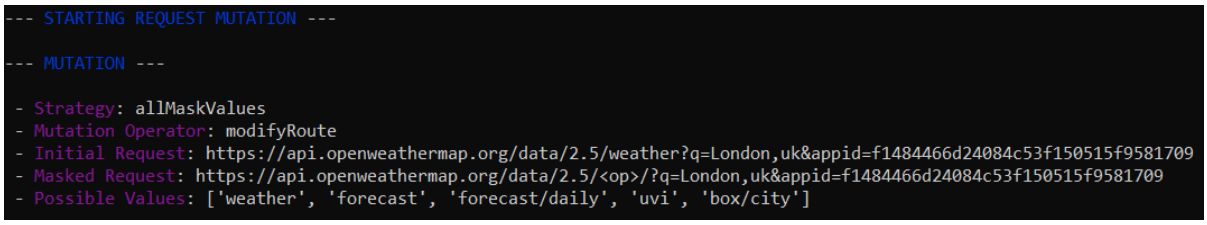


Figure 25: Example of the mutation interface displayed by *MutGPT*.

Figure 25 displays the first mutation of the program. Initially, the title `STARTING REQUEST MUTATION` is displayed to warn that the request mutation process will begin. Then, a first mutation begins; Relevant information and results are displayed in the command prompt. Each displayed element represents a certain information w.r.t. the current mutation:

- **Strategy**: The utilized strategy for the current mutation. As an initial request mutation strategy and an improved strategy have been presented, the program is thus able to execute one or the other. Here, the strategy name is `allMaskValues`, which represents the Multiple Request Mutation Strategy. However, it is also possible to execute the Single Request Mutation Strategy, entitiled `singleMaskValues`. As the MRMS displays better performance, it is enabled by default.
- **Mutation Operator**: The name of the mutation operator that has been selected for the current mutation, as presented in Section 4.2. In the demonstration figure, the `modifyRoute` mutation operator is selected, which consists in modifying an existing route of the request.
- **Initial Request**: The request seed that has been selected for the current mutation. It is entitled "initial" as it represents the base structure of the request, without any mask or mutation applied onto it.
- **Masked Request**: The request seed with the chosen mutation operator applied onto it. As described in Section 4.4, the masked request can either contain an `<op>` token mask or directly consist in the mutated request. For this example, the `modifyRoute` mutation operator is applied onto the initial request, which results in the `weather` route of the request being replaced with an `<op>` token mask.

- **Possible Values**: List of all possible values to replace the mask that have been found by the Large Language Model. Each value will replace the `<op>` token mask of the masked request in order to generate mutated requests.

Verifying the Validity of the Mutated Requests. As a result of generating mutated requests, the validity of each mutated request needs to be verified. In order to do so, *MutGPT* sends each mutated request to the API endpoint and analyzes the server response.

```
-> Verifying request validity: https://api.openweathermap.org/data/2.5/forecast/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 200, 'validity': 'validRequest', 'title': 'No Title Found', 'content': '{"cod":"200","message":0,"cnt"...'}
-> New route found: /data/2.5/forecast/
-> New parameter found: q
-> New value found for parameter 'q': London,uk
-> New parameter found: appid
-> New value found for parameter 'appid': f1484466d24084c53f150515f9581709
-> Added in seeds
```

Figure 26: Example of a valid mutation interface displayed by *MutGPT*.

Figure 26 illustrates an example of the displayed interface for a valid mutated request. First, a blue title `Verifying request validity` indicates the mutated request for which the validity is verified. Next, the HTTP response information from the server is displayed. This information contains:

- **Status**: The status code returned by the server for the mutated request. In this case, the status code is `200`, which indicates a successful request.
- **Validity**: The validity group of the mutated request. To recall, validity groups are detailed in Section 4.3.2. The value `validRequest` in the example indicates that the mutated request is contained in the 200-299 status code range, which renders the request valid.
- **Title**: The title of the response data returned by the server. As detailed in Section 4.3.3 and Section 4.3.4, a title is not always available as most APIs only return a JSON object containing the requested data, and not a HTML page. For the current example, the title is `"No Title Found"`, which indicates that the returned data did not contain a `<title>` HTML tag.
- **Content**: The data contained in the server HTTP response. As said in the bullet point above, Section 4.3.4 details that APIs often respond to valid requests with a JSON object containing the requested data. In this case, the content is indeed a JSON object, however its display has been voluntarily shortened by *MutGPT* for readability purposes. In the figure, the object contains a `"cod": "200"` parameter, indicating the user that the mutated request is indubitably a valid request as confirmed by the server.

Under the server response information, green text preceded by arrows indicates that the valid mutated request is being analyzed in order to further infer the API specification. In this example, as the initial `/weather` route of the request has been replaced with the `/forecast` route by mutation, the newly discovered route has been added to the API specification. Since the route was not yet contained in the specification, all parameters of the mutated request have also been inferred. Consequently, merely modifying the route of the request seed with a new route allowed the program to discover an unexplored route, capable of containing the same parameters as the previous route.

Finally, another green text preceded by an arrow indicates the user that the mutated request has been added to the seed list, as it was not yet contained inside of it.

```
-> Verifying request validity: https://api.openweathermap.org/data/2.5/forecast/daily/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 401, 'validity': 'invalidRequest', 'title': 'No Title Found', 'content': '{"cod":401, "message": "Invali...'}
-> Rejected in seeds
```

Figure 27: Example of an invalid mutation interface displayed by *MutGPT*.

However, mutated requests can also be invalid. Figure 27 exhibits the interface displayed when a mutated request is invalid. As presented, the invalid request is rejected from the seed list, thus resulting in an unsuccessful mutation.

Furthermore, *MutGPT* can also warn the user for potential in-page errors, as described in Section 4.4. Figure 28 displays an example of such warning message displayed in the interface. As shown in the figure, the

```
-> Verifying request validity: https://api.helioviewer.org/v2/layers/
- Validity: {'status': 200, 'validity': 'validRequest', 'title': 'Helioviewer.org API - Error', 'content': 'No Content Found'}
-> Detected a potential in-page error
```

Figure 28: Example of a warning displayed by *MutGPT* for a potential in-page error detected.

status code of the response is `200`, however the parsed title displays `"Helioviewer.org API - Error"`, which was correctly detected by *MutGPT* as an in-page error. A request for the *Helioviewer* API is used solely for this example, as Section 4.3.1 details that the API displays in-page errors for invalid requests.

```
--- MUTATION ---

- Strategy: allMaskValues
- Mutation Operator: modifyRoute
- Initial Request: https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Masked Request: https://api.openweathermap.org/data/2.5/<op>/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Possible Values: ['weather', 'forecast', 'forecast/daily', 'uvi', 'box/city']

-> Verifying request validity: https://api.openweathermap.org/data/2.5/weather/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 200, 'validity': 'validRequest', 'title': 'No Title Found', 'content': '{"coord":{"lon":-0.1257,"lat":...'}
-> Added in seeds

-> Verifying request validity: https://api.openweathermap.org/data/2.5/forecast/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 200, 'validity': 'validRequest', 'title': 'No Title Found', 'content': '{"cod":"200","message":0,"cnt"...'}
-> New route found: /data/2.5/forecast/
-> New parameter found: q
-> New value found for parameter 'q': London,uk
-> New parameter found: appid
-> New value found for parameter 'appid': f1484466d24084c53f150515f9581709
-> Added in seeds

-> Verifying request validity: https://api.openweathermap.org/data/2.5/forecast/daily/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 401, 'validity': 'invalidRequest', 'title': 'No Title Found', 'content': '{"cod":401, "message": "Invali...'}
-> Rejected in seeds

-> Verifying request validity: https://api.openweathermap.org/data/2.5/uvi/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 404, 'validity': 'invalidRequest', 'title': 'No Title Found', 'content': '{"message":"Not Found"}'}
-> Rejected in seeds

-> Verifying request validity: https://api.openweathermap.org/data/2.5/box/city/?q=London,uk&appid=f1484466d24084c53f150515f9581709
- Validity: {'status': 404, 'validity': 'invalidRequest', 'title': 'No Title Found', 'content': '{"cod":"404","message":"Intern...'}
-> Rejected in seeds

-> Progress: 1/10
```

Figure 29: Example of a complete mutation process of *MutGPT*.

To conclude the current demonstration step, Figure 29 presents the complete mutation process that has been explained throughout the step. At the end of the mutation, a yellow `Progress` arrow is displayed, indicating the number of the current mutation out of the total number of required mutations.

Continuation of the Mutation Process. As the first mutation has been completed, 9 other mutations remain. The previous step is then repeated 9 times, with new random request seeds and mutation operators.

End of the Mutation Process. When all mutations have been executed, the program terminates. A summary of the execution is displayed, containing all status codes obtained and their occurrences. The number of discovered routes, parameter names and parameter values is also displayed. The overview allows the program user to identify if *MutGPT* was able to correctly generate mutated requests. Figure 30 displays an example of the execution results.

```
- Status Code Results: {200: 19, 400: 3, 404: 1}
- Found Routes: 3
- Found Parameter Names: 7
- Found Parameter Values: 26
```

Figure 30: Example of the execution summary displayed by *MutGPT*.

Program Outputs. As the core goal of the tool is to automatically infer API specifications, *MutGPT* stores

established results in files. By going into the main folder of *MutGPT*, a `output` folder contains 3 sub-folders, themselves containing uncovered API data:

- `bases`: Contains the initial API information uncovered by the program in a JSON object. This information consists in the API base URL, the API documentation URL and the number of base routes. This information is important to save, in order to avoid re-prompting the model if the user undergoes another mutation process after terminating the program. Figure 31 displays an example of the initial API information uncovered for the *OpenWeatherMap* API.
- `grammars`: Contains the inferred specification of the API in a JSON object. The structure of such object is described in Listing 5. Figure 32 displays an example of the inferred specification for the *OpenWeatherMap* API.
- `seeds`: Contains the request seeds, corresponding to all valid initial requests and all valid mutated request. The requests are separated by new line characters, in order for the program to open the file and separate the request seeds effortlessly into a *Python* list. Figure 33 displays an example of the seed list for the *OpenWeatherMap* API.

As multiple APIs can be used with *MutGPT*, each previously detailed folder accommodates the information of all utilized APIs in different files. In order to distinguish data, each file is named after the name of the API, followed by either `_base.txt`, `_grammar.txt` or `_seeds.txt`, depending on the folder. For instance, the file containing the seed list of the *OpenWeatherMap* API is named `OpenWeatherMap_API_seeds.txt`, and is found in the `outputs/seeds` folder.


```
{
    "doc": "https://openweathermap.org/api",
    "base": "https://api.openweathermap.org/data/2.5/",
    "nbBaseRoutes": 2
}
```


Figure 31: Example of the content found in the `bases` folder of *MutGPT* for the *OpenWeatherMap* API.


```
{
    "https://api.openweathermap.org": {
        "/data/2.5/weather/": {
            "q": [
                "London,uk",
                "London"
            ],
            "appid": [
                "f1484466d24084c53f150515f9581709"
            ],
            "zip": [
                "94040,us"
            ]
        }
    }
}
```


Figure 32: Example of the content found in the `grammars` folder of *MutGPT* for the *OpenWeatherMap* API.


```
https://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=f1484466d24084c53f150515f9581709
https://api.openweathermap.org/data/2.5/weather/?q=London&appid=f1484466d24084c53f150515f9581709
https://api.openweathermap.org/data/2.5/weather/?zip=94040,us&appid=f1484466d24084c53f150515f9581709
```


Figure 33: Example of the content found in the `seeds` folder of *MutGPT* for the *OpenWeatherMap* API.

5 Evaluation and Results

For this section, the **Research Questions** are initially presented. Next, for each research question detailed, a section is presented. Each section details the setup, metrics and results used in experiments for the described research questions. Finally, a **Threats to Validity** section is exposed, which consists in describing the encountered internal and external validity threats to which the work is subject.

Regarding the availability of the evaluation and results, a `/outputs/test` folder can be found on the *GitHub* repository of *MutGPT* at the following URL: `https://github.com/alixdecr/MutGPT/tree/master/outputs/tests`. The folder contains a sub-folder for each research question, themselves containing the results of the experiments.

5.1 Research Questions

This section describes the research questions which were explored for the purpose of leveraging Large Language Models to automatically infer API specifications.

RQ1: *What is the impact of the prompt temperature parameter on the Large Language Model predictions, in terms of request validity and diversity?*

RQ2: *Is MutGPT more performing than the preliminary strategies in terms of generating valid and diverse requests?*

RQ3: *Can MutGPT infer the documentation of APIs, and possibly discover undocumented features?*

For each research question, results will only cover HTTP requests of the GET method, as it is the most used method for API requests. Other methods such as POST or PUT are not considered in the scope of this work.

5.2 RQ1: Impact of the Temperature Parameter on Model Responses

Before comparing *MutGPT* with the other explored strategies, an initial research question consists in analyzing the impact of the prompt temperature parameter on the leveraged model. As presented in Section 3.1.2, it is possible to vary the *temperature* of the model in the prompt parameters, which consists in controlling the randomness of responses. The lower the temperature, the more deterministic and repetitive the model is supposed to be. Thus, for the purpose of generating valid and diverse requests, the impact of this parameter is explored.

5.2.1 Setup

In order to analyze the impact of the temperature prompt on the model response, the Basic Request Prompt Strategy is used for 2 different APIs: the *GBIF Species* API and the *OpenWeatherMap* API. For each API, 10 requests are prompted to the model. To observe the temperature variation, this process is repeated 3 times for a temperature parameter of 0 - no randomness, 0.7 - the default *GPT* randomness - and 2 - the maximum randomness. Moreover, the *text-davinci-003* model is leveraged for the experiment.

5.2.2 Metrics

In order to formally analyze and define the obtained temperature results, the following metrics are employed:

- **Number of Valid Requests**: The number of generated requests that are valid.
- **Number of Different Routes**: The number of different routes found in all valid requests.
- **Number of Different Parameter Names**: The number of different parameter names found in all valid requests.
- **Number of Different Parameter Values**: The number of different parameter values found in all valid requests.

Temperature	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
0	**10**	**1**	1	**10**
0.7	**10**	**1**	**9**	**10**
2	0	0	0	0

Table 1: Results obtained for 10 requests to the *GBIF Species* API with the Basic Request Prompt Strategy for a temperature parameter of 0, 0.7 and 2. The best results are colored in green.

5.2.3 Results

In order to display obtained results, Table 1 and Table 2 present the outcomes of the request generation for both APIs, with the 3 specified temperature amounts. As shown, a temperature of 0.7 exhibits the best results, uncovering 19 valid requests, 4 different routes, 18 different parameter names and 22 different parameter values for both APIs. The temperature amount of 0 is slightly behind, uncovering 18 valid requests, 3 different routes, 7 different parameter names and 21 different parameter values. For the *GBIF Species* API, the temperature 0 only uncovered a single parameter name.

However, model responses with the maximum temperature amount of 2 did not result in any valid requests. By observing the results, it is apparent that such temperature exhibits extreme randomness, resulting in responses not being requests. To illustrate, the response returned for the *GBIF Species* API is the following:

```
[‘‘GET rest download FirstRecord points Antarctica coordinates tile BOX countries Holland
Pro crack Installatin macrobot UsYouzip URL Updates SatelliteAbout newBlog protocol PHPastem
body te section object states CompleteStatuts header definitions realis XP 200 Column 38954
ge 15 Found 9...]
```

And the response returned for the *OpenWeatherMap* API:

```
["wester bipping 420km presic suplan! glee ridimet ip by replairrim duxt = pieying cig
swoffonaineed KEAL czese1 tv/jletefast", 23140 shaya pleatic A Noityellapi hyp by env piHTpit
PEBatethie li1 mations queryvarxfkeyarmita cor M coptempohraphansingsissot formade hiELHEe
Get/?aptarsqi 34ALLCTYuTRv ob In ther...]
```

As a maximum temperature causes the model to completely diverge from the prompted task, another experimentation would consist in verifying at which temperature amount does the model generate structurally incorrect requests. In order to do so, a small-scaled manual experiment was conducted on the *OpenAI Playground* page, prompting the model with the same task for temperatures above 0.7 and below 2. By testing multiple temperatures, results demonstrated that a temperature of 1 already begins to display invalid behaviors. Figure 34 displays an example of a response for a temperature value of 1. As shown, the model returns incorrect routes for the *GBIF Species* API such as `/GPA, /GMBE` and `/GBAYE`, resulting in worse outcomes than the ones observed for a temperature of 0 or 0.7.

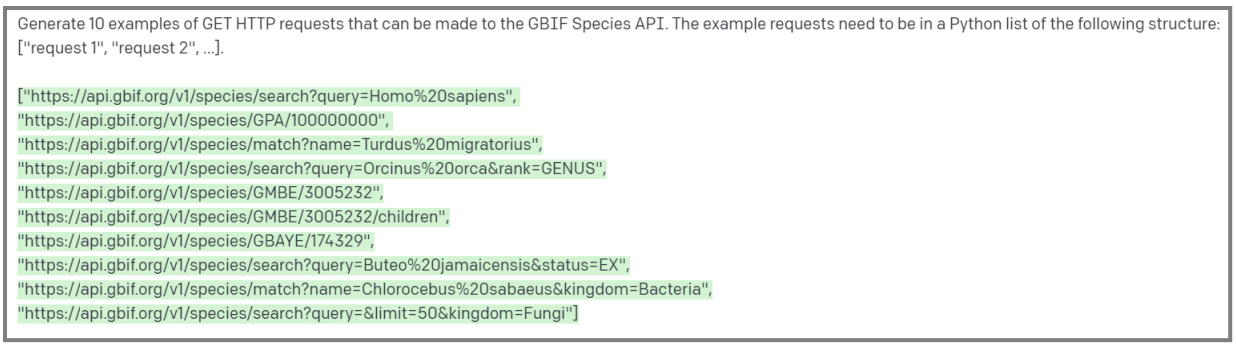


Figure 34: Example of requests generated by the *text-davinci-003* model with a temperature of 1.

Temperature	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
0	8	2	6	11
0.7	**9**	**3**	**9**	**12**
2	0	0	0	0

Table 2: Results obtained for 10 requests to the *OpenWeatherMap* API with the Basic Request Prompt Strategy for a temperature parameter of 0, 0.7 and 2. The best results are colored in green.

Overview: Research Question 1

In retrospect, the temperature value of 2 displayed the worst performance, not being able to generate structured requests. Thus, such temperature is not considered, nor other temperatures above 0.7 as too much randomness could cause issues with the validity of requests. While a temperature value of 0 displayed correct results, the deterministic and repetitive behavior can cause issues with the generation of varied requests, as it will attempt to generate similar requests. Consequently, as the temperature value of 0.7 displayed superior results, it will be chosen by default in all strategies for the next research questions. Indeed, such temperature is an ideal equilibrium for validly structured requests containing diversified routes and parameters.

5.3 RQ2: Strategy Comparison

Another research question consists in verifying that the implemented tool, *MutGPT*, is more performing than the preliminary strategies that were explored. To recall, the considered preliminary strategies consist in the Basic Request Prompt Strategy and the Complex Request Prompt Strategy. As *MutGPT* consists of an initial Single Request Mutation Strategy and an improved Multiple Request Mutation Strategy, both strategies are also compared for the evaluation. The current evaluation does not aim to analyze the performance of API specification inference, but rather to compare strategies in terms of API requests validity and diversity.

5.3.1 Setup

For this experimentation, 4 existing APIs are chosen, some of which have already been described in the previous sections of the document: the *GBIF Species* API, the *OpenWeatherMap* API, the *MusicBrainz* API [55] and the *REST Countries* API [70].

For each request generation strategy, an amount of 20 requests - or mutations depending on the strategy - is specified as input to the relevant program. To recall, the evaluated strategies consist in the following: the Basic Request Prompt Strategy, the Complex Request Prompt Strategy, the Single Request Mutation Strategy and the Multiple Request Mutation Strategy.

Moreover, each prompt is sent to the *text-davinci-003 OpenAI* model, as it proved to be able to correctly generate requests for each strategy.

5.3.2 Metrics

In order to formally analyze and define the obtained results, the following metrics are employed:

- **Number of Valid Requests**: The number of generated requests that are valid.
- **Number of Different Routes**: The number of different routes found in all valid requests.
- **Number of Different Parameter Names**: The number of different parameter names found in all valid requests.

- **Number of Different Parameter Values**: The number of different parameter values found in all valid requests.

If a generated request is invalid, the content of the request is ignored and not analyzed. Thus, such requests cannot result in the discovery of new routes and parameters.

Regarding the number of different parameter names metric, a parameter name is considered different from another parameter name if their name strings are not identical. This signifies that if the strategy uncovered a parameter name that can be used in 2 different routes, the parameter number will still count as 1. For instance, if the strategy generates `/route1?param` and `/route2?param`, `param` will only add 1 to the total number of different parameter names.

Moreover, for the number of different parameter values metric, this amount represents the number of unique values uncovered for each parameter name. To illustrate, if 2 parameters `param1=value` and `param2=value` are generated, `value` will count as 2 parameter values as its occurrence is found in 2 different parameter names. However, for 2 parameters `param1=value` and `param1=value`, `value` will only count as 1 new value, as the values are duplicated for the same parameter name.

As the response time of the *OpenAI* model can vary depending on the current server load, it is not considered as a metric in order to avoid biased results. However, for 20 requests, each strategy did not exceed a total model response time of 1 minute.

5.3.3 Results

Regarding obtained results, the outcomes of each strategy are represented in different tables containing the API names and the various metrics described above. Along with each table, an overview of the observed results is given with additional comments regarding the in-depth analysis of generated requests.

Basic Request Prompt Strategy. Table 3 presents the obtained results for the initial request prompt strategy. As displayed, the strategy was able to generate 38 total valid requests out of the 4x20=80 initially prompted amount, corresponding to a 47.5% request validity rate. The strategy was able to uncover 10 different routes and 16 different parameter names with 50 corresponding parameter values.

Interestingly enough, by analyzing the invalid responses generated for the *MusicBrainz* API, the 7 last requests generated `503 Service Unavailable` status codes. This status code is returned whenever the corresponding server is down for maintenance or that it is overloaded. As the *MusicBrainz* API was online at the time, the cause of the status code is that the server was overloaded with the sent GET requests. As described in Section 4.3.2, such status codes are rarely returned by APIs as it divulges potentially dangerous and exploitable server errors. However, the behavior is deliberate, as described on the *MusicBrainz* API rate limiting page [56]. Thus, such issue does not need to be reported.

For the *REST Countries* API, the table displays that all generated requests are invalid, thus not discovering any routes or parameters. By analyzing the response in-depth, the uncovered issue is that while all generated requests appeared valid, each one contained an incorrect top-level domain name. Indeed, while the adequate API endpoint of the *REST Countries* API is `https://restcountries.com`, the model generated requests to `https://restcountries.eu`, rendering the requests invalid. This issue is due to the fact that the leveraged *text-davinci-003* model is trained up to June 2021, as of May 2023. However, the API endpoint of the *REST Countries* API has been modified [23] after this time by the previously described URL. In conclusion, the initially given API base URL was valid at the time of the model's training data limit.

Complex Request Prompt Strategy. Table 4 presents the obtained results for the Complex Request Prompt Strategy. As shown, the strategy was able to generate 49 valid requests out of 80 in total, corresponding to a 61.25% request validity rate. 11 different routes, 34 different parameter names and 75 different parameter values were also uncovered.

A lot of parameter values were uncovered for the *MusicBrainz* API. By analyzing the generated requests, 2 parameter names, `limit` and `offset`, contain a lot of different integer parameter values. While this is not a issue for the experiment, the discovery of an excessive amount of integer parameter values does not bring any

API Name	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
GBIF Species	**20**	1	3	**20**
OpenWeatherMap	6	3	**8**	13
MusicBrainz	12	**6**	5	17
REST Countries	0	0	0	0
TOTAL	**38/80**	**10**	**16**	**50**

Table 3: Results obtained with the Basic Request Prompt Strategy for 20 prompted requests. The best result for each metric category is displayed in green.

API Name	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
GBIF Species	**20**	1	**20**	27
OpenWeatherMap	14	4	10	18
MusicBrainz	15	**6**	4	**30**
REST Countries	0	0	0	0
TOTAL	**49/80**	**11**	**34**	**75**

Table 4: Results obtained with the Complex Request Prompt Strategy for 20 prompted requests. The best result for each metric category is displayed in green.

new information regarding the corresponding parameter[16].

Similarly to the Basic Request Prompt Strategy results, the *REST Countries* API resulted in all invalid requests as the high-level domain `.eu` is incorrect for the API endpoint. However, even when replacing the domain with `.com`, all requests remain invalid as the API route is incorrectly written by the model nonetheless.

Single Request Mutation Strategy. Table 5 presents the obtained results for the Single Request Mutation Strategy. The strategy was able to generate 68 valid requests out of 80 mutations, starting from a single automatically found seed by the model for each API. This amount corresponds to a 85% request validity rate. The strategy was able to discover 10 routes, 28 parameter names and 56 parameter values.

As the model found an invalid *REST Countries* API base URL similarly to the previous strategies, the correct base URL was manually given as input to the program. This URL served as an initial seed, from which mutations could expand on. While the base URL is not an adequate seed, the strategy was still able to discover 14 valid requests containing 3 different routes and parameter names along with 10 different parameter values.

However, when comparing results with the Complex Request Prompt Strategy, the Single Request Mutation Strategy seems to be underperforming. While the current strategy achieves a higher request validity rate on account of the manually given *REST Countries* API base URL, the previous strategy demonstrated more diversity in the routes and parameters found. Even though the tables are displayed for strategy comparison purposes, prompting the model for a certain amount of request versus mutating a single request seed is not an identical process. While prompting the model for requests is advantageous to get diverse initial seeds, expanding the API specification requires a mutation process to potentially discover slight variations that could exist in the API. As presented in Section 3.1.3, the *text-davinci-003* model contains 2 limitations that support the use of the mutation process. These limitations consist in the cost of using the model and the token limit in the response, which would hinder the prompting for large amounts of requests. Regardless, both strategies proved to be able to correctly generate API requests in an unassisted manner for 3 out of 4 tested APIs.

[16]If a parameter consists of integer values, prompting the model to explore *boundary values* [13] such as a negative integer or 0 could be interesting for testing purposes.

API Name	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
GBIF Species	**20**	**3**	7	15
OpenWeatherMap	14	2	**10**	14
MusicBrainz	**20**	2	8	**17**
REST Countries	14	**3**	3	10
TOTAL	**68/80**	**10**	**28**	**56**

Table 5: Results obtained with the Single Request Mutation Strategy for 20 mutations. The best result for each metric category is displayed in green.

Multiple Request Mutation Strategy. Table 6 presents the obtained results for the Multiple Request Mutation Strategy. As the strategy is able to generate multiple mutations per model prompt, a total of 327 requests were valid out of 520 generated requests. The results were obtained with a total of 80 model prompts, corresponding to the same amount as the other strategies. The strategy achieved a request validity rate of 62,88% and generated 6.5x more requests than the other strategies, for the same amount of model prompts. 18 different routes, 73 different parameter names and 177 different parameter values were discovered, considering all 4 APIs.

First, the request validity rate is lower compared to other strategies. By analyzing undergoing mutations in the tool, this reduction is mainly due to the `removeRoute, modifyRoute and addRoute` mutation operators. Even though such operators are of foremost importance to discover new API routes, generated routes by the model do not always result in valid requests. Indeed, as such mutations consist in only modifying a route of the request and not the request parameters, it is possible for the newly found route to not contain the described parameters, resulting in an invalid request. Moreover, as certain APIs have a maximum route depth, the `addRoute` mutation operator often results in only invalid requests if the API contains no further routes. For instance, if the model returned 10 possible routes for the `addRoute` mutation operator and all of the resulting mutated requests are invalid, this directly adds 10 to the total amount of requests, reducing the request validity rate.

Regarding the *GBIF Species* API, results demonstrated that over 41 different parameter names were uncovered, which represents 56.16% of all uncovered parameter names for the experiment. As this amount seemed excessive, the official documentation of the *GBIF Species* API was compared to the obtained parameter results. *MutGPT* was able to generate 19/27 officially documented parameters of the *GBIF Species* API, resulting in a 70.37% parameter discovery rate. Moreover, 22 parameter names have been found by the strategy and are not documented on the API website. This additional number of parameter names is due to 1 of 2 reasons:

- The parameters exist for the *GBIF Species* API, but are not documented on the official website.
- The parameters do not exist for the API, and consist in false positives contained in valid requests.

The latter reason happens when the API server ignores certain parameters of a request when treating it. Indeed, if a sent request contains a valid route and a couple of valid parameters followed by an invalid parameter, it is possible for the last parameter to be ignored by the API server.

Example: Request Parameter as a False Positive

The following request containing an invalid `fake=0` parameter is sent to the *GBIF Species* API:

Sent Request: `https://api.gbif.org/v1/species/match?name=Homo+sapiens&fake=0`

The server will then reply with a `200 OK` status code, as the first part of the request `/match?name=Homo+sapiens` is valid. The server returns data corresponding to the valid part of the request and ignores the invalid `fake=0` parameter, resulting in a false positive for this parameter.

API Name	Nb Valid Requests	Nb Different Routes	Nb Different Parameters Names	Nb Different Parameter Values
GBIF Species	**104/165**	5	**41**	51
OpenWeatherMap	80/101	1	9	39
MusicBrainz	79/99	**10**	10	**52**
REST Countries	64/155	2	13	35
TOTAL	**327/520**	**18**	**73**	**177**

Table 6: Results obtained with the Multiple Request Mutation Strategy for 20 mutations. The best result for each metric category is displayed in green.

The treatment of predominantly valid requests containing invalid parameters differs depending on the API in use. As invalid parameters can be ignored by certain API servers, it is difficult to detect such false positives, and mainly depends on an adequate API implementation. Thus, it is possible for the leveraged Large Language Model to generate invalid parameters in few cases. As the leveraged model is *text-davinci-003*, requests containing invalid parameters might result from:

- The training data of the model - which is up to June 2021, resulting in the model returning parameters that do not exist in the API anymore as of 2023.
- The specified temperature of the model, which controls the randomness of the generated outputs.

Regarding the temperature of the model, experimentation with the *GBIF Species* API demonstrated that modifying the specified temperature of the model between 0 and 0.7 had no impact on returned parameter names. As a matter of fact, both temperatures resulted in similar uncovered parameter names for the API.

Overview: Research Question 2

To conclude, results demonstrated that the Multiple Request Mutation Strategy employed by *MutGPT*[a] is the best strategy for generating valid and diversified API requests. The strategy was able to generate 6.5 times more requests compared to the other strategies, and with an identical amount of model prompts. Furthermore, the MRMS utilizes less response tokens than other strategies, as it only returns mask values of given requests and not whole HTTP requests. The experimentation also uncovered that request parameters can consist in false positives. As requests usually require certain combinations of parameters in order to be valid, the described issue only resulted in a few false positives for the *GBIF Species* API. The generated specifications of the other APIs did not result in falsely inferred invalid parameters.

[a]The Multiple Request Mutation Strategy is considered as the default strategy employed by *MutGPT*, even though the Single Request Mutation Strategy is also available with the tool.

5.4 RQ3: Inferring API Specifications with *MutGPT*

As Section 5.3 demonstrated that the Multiple Request Mutation Strategy utilized by *MutGPT* proved to be the best strategy for the purpose of generating valid and diverse requests, a third research question consists in verifying the capabilities of *MutGPT* to infer API specifications. The results found for this research question will evaluate the initial research idea, which consisted in verifying if it is possible to automatically infer API specifications by leveraging Large Language Models.

5.4.1 Setup

For this experimentation, *MutGPT* is executed for 3 different APIs. The chosen APIs are the *GBIF Species* API, the *Random User Generator* API and the *Deezer* API [44]. For each API, obtained results after 20

MutGPT iterations are presented, similarly to the previous research question. Afterwards, 20 more iterations are executed - containing the already inferred data from the previous iterations - and additional results are also detailed.

5.4.2 Metrics

As the purpose of the current experimentation is to infer API specifications, the utilized metrics consist in:

- **Number of Valid Requests / Number of Generated Requests (VR / GR)**: The number of generated requests that are valid compared to the total number of generated requests by *MutGPT*.
- **Number of Found Routes / Number of Documented Routes (FR / DR)**: The number of different routes found by *MutGPT* in all valid requests compared to the total number of routes documented on the official API website.
- **Number of Found Parameters / Number of Documented Parameters (FP / DP)**: The number of different parameters found by *MutGPT* in all valid requests compared to the total number of parameters documented on the official API website.
- **Number of Different Parameter Values (DPV)**: The number of different parameter values found in all valid requests.
- **Number of Undocumented Found Routes (UFR)**: The number of different routes found by *MutGPT* that are not documented on the official API website.
- **Number of Undocumented Found Parameters (UFP)**: The number of different parameters found by *MutGPT* that are not documented on the official API website.

The UFR metric reflects undocumented routes that are truly valid, as an invalid route in a API request will always lead to an error. However, the UFP metric can reflect valid requests that contained invalid parameters, as explained in the example in Section 5.3.3.

5.4.3 Results

The following paragraphs detail the obtained results of the experiment. For each API, a table is presented, containing the utilized metrics and the corresponding results. Additional clarifications w.r.t. the results are also given.

***GBIF Species* API**. For this API, the following official API documentation URL is used: `https://www.gbif.org/developer/species`. Table 7 displays the obtained results for the *GBIF Species* API. As presented, the first 20 iterations of *MutGPT* discovered 5/5 documented routes and 18/27 documented parameters. 192 requests were generated, 172 requests being valid. The iteration also uncovered 115 different parameter values, 1 undocumented route and 21 undocumented parameters.

Regarding the second iteration process, 161 additional requests were valid out of 178, resulting in a total of 370 requests generated. As a total of 333 requests were valid, the request validity rate is 90%. Moreover, 3 new documented parameters were discovered during the second batch of iterations, bringing the total amount of found parameters to 21. The second execution also uncovered 94 new parameter values and 18 undocumented parameters were found. Thus, the 2 executions achieved a 100% route discovery rate and a 77.77% parameter discovery rate.

The specified amount of documented route only concerns the routes used to find the name usages of species. However, the API also contains 15 additional routes requiring the specification of a species ID. For instance, the *GBIF Species* API contains a `/species/{int}` route and a `/species/{int}/parents` route, which would require the tool to modify route values and not simply replace them.

Interestingly enough, *MutGPT* was able to find an undocumented route of the *GBIF Species* API. Indeed, the tool discovered the `/species/lookup` route of the API, which is not referenced on the website documentation. Such finding has been notified to the developers of the API, and proves that the leveraged *text-davinci-003* model is able to discover undocumented API routes with its training data.

Iterations	VR / GR	FR / DR	FP / DP	DPV	UFR	UFP
20	172/192	5/5	18/27	115	1	21
40 (+ 20)	333/370 (+ 161/178)	5/5 (+ 0)	21/27 (+ 3)	209 (+ 94)	1 (+ 0)	39 (+ 18)
Percentage	90%	100%	77.77%			

Table 7: Cumulative results obtained for 20 and 40 iterations by *MutGPT* for the *GBIF Species* API. Percentages of the first 3 metrics are displayed below.

VR / GR	FR / DR	FR / DR Percentage	UFR
58/182	11/15	73.33%	1

Table 8: Results obtained for 20 iterations by *MutGPT* to target specific route of the *GBIF Species* API, using only mutation operators relative to discovering routes and with a given seed.

While *MutGPT* is capable of discovery valid and undocumented routes, it also discovered 41 undocumented parameters. In all likelihood, such parameters consist in false positives. To illustrate, 2 undocumented parameters `speciesKey` and `familyKey` were found by the model. These values cannot be inserted as parameters in a requests, however they can exist in the JSON response data returned by the *GBIF Species* API server. Indeed, by analyzing response data for valid requests, `speciesKey` and `familyKey` are sometimes used as keys in the said data. Presumably, the Large Language Model analyzed such data during training, and misinterpreted it as potential request parameters for the *GBIF Species* API.

In order to verify if the model is able to find the 15 additional routes of the API described in the previous paragraphs, an in-between experiment was conducted. For this small-scaled experiment, the following seed was given to the program: `https://api.gbif.org/v1/species/10`, with a real value for the placeholder `/{id}`. In order to quickly detect if *MutGPT* would be able to discover the routes, only `addRoute, removeRoute` and `modifyRoutes` were used as mutation operators. Moreover, 20 iterations were executed.

Table 8 displays the obtained results. As shown, *MutGPT* was able to uncover 11/15 routes, resulting in a 73.33% route discovery percentage. Only 31.87% of the generated requests were valid, as the model generated various requests containing invalid routes of the API. However, another undocumented route was discovered: `/species/{id}/identifier`. With further analysis, the discovered route is truly a valid and undocumented route, as the *GBIF Species* API returns status codes in the 400 range when a route is invalid. In fact, a request containing `/species/{id}/identifier` returned a `200 OK` status code, while a request containing `/species/{id}/fakeroute` returned a `404 Not Found` status code. This behavior signifies that the `/species/{id}/synonyms` route is implemented, however it returns empty data in the JSON response, hence its absence from the API documentation.

***Random User Generator* API**. For this API, the following official API documentation URL is used: `https://randomuser.me/documentation`. Table 9 displays the obtained results for the *Random User Generator* API. As shown, the first 20 iterations of the program were able to uncover 5/6 routes and 9/10 parameters documented on the API website, with 152 valid requests out of 169 generated in total. The first iteration also uncovered 84 different parameter values. The next 20 iterations did not find any new routes or parameters, and generated 124 new valid requests out of 127 generated in total, bringing the total to 276/296 valid requests. Thus, generated requests for the *Random User Generator* API achieved a record of 93.24% validity rate. Moreover, the following 20 iterations uncovered 29 new parameter values and 2 undocumented parameters.

Iterations	VR / GR	FR / DR	FP / DP	DPV	UFR	UFP
20	152/169	5/6	9/10	84	0	0
40 (+ 20)	276/296 (+ 124/127)	5/6 (+ 0)	9/10 (+ 0)	113 (+ 29)	0 (+ 0)	2 (+ 2)
Percentage	93.24%	83.33%	90%			

Table 9: Cumulative results obtained for 20 and 40 iterations by *MutGPT* for the *Random User Generator* API. Percentages of the first 3 metrics are displayed below.

The experimentation demonstrates that with only 20 iterations, more than 80% of the routes and 90% of the parameters of the *Random User Generator* API documentation were found by *MutGPT*. For this particular API, adding 20 more iterations did not result in the discovery of the last route and the last parameter of the API. Additionally, 29 new parameter values were found in the second execution.

However, the following 20 iterations found 2 undocumented parameters. By inspecting the parameters, their names correspond to `legacy` and `reg`, which are not found in the API documentation. With further investigation, such parameters are presumably false positives, and not truly undocumented parameters.

***Deezer* API**. *Deezer* is a music streaming app similar to *Spotify*. Its API [44] allows developers to retrieve information relative to albums, artists, playlists, etc. For this API, the following official API documentation URL is used: `https://developers.deezer.com/api`. As the *Deezer* API is larger than the previously tested APIs, the evaluation process differs. Instead of attempting to find all API routes and parameters with *MutGPT*, the evaluation will consist of 2 experiments.

The first experiment consists in finding the most amount of documented parameters in different API routes. To do so, 3 routes of the *Deezer* API are selected: the `/album` route, the `/artist` route and the `/track` route. A basic seed is given to *MutGPT* for each route, in order to begin the mutation process in the required route. Moreover, the `addRoute, removeRoute, modifyRoute, removeParameter` and `modifyParameterValue` mutation operators are temporarily removed. This allows *MutGPT* to focus on discovering parameters only. Each attempt at discovering route parameters will be executed in 20 iterations only.

Table 10 displays the results obtained for the first experimentation of the *Deezer* API. As shown, a total of 42/73 documented parameters were found, resulting in a 57.53% parameter discovery rate.

First, the table column containing the valid requests displays that all generated requests were valid. However, by investigating data returned from the *Deezer* API server, a lot of generated requests are in reality invalid, as the returned data only contained an `"error"` JSON object. As the structure of response data can vary from an API to another, it is difficult to automatically detect exactly if response data corresponds to a valid response or if it is to warn the user that the sent request is invalid. Thus, even though the tool was able to generate requests containing valid API parameters, the request validity rate is not 100%.

Moreover, the parameter discovery rate is lower than the 2 previouly tested APIs. While *MutGPT* uncovered 80% parameters of the `/artist` route, the percentage decreases to 63.33% for the `/album` route and 39.28% for the `track` route. With further analysis, the issue is partially due to to the training data limit of the *text-davinci-003* model. As it is limited to June 2021, it is highly possible that the *Deezer* API has since updated its parameter usage and the relevant documentation that was inspected for this experiment. This suggestion cannot be proved, as no information was found regarding the update history of the *Deezer* API.

The second experiment aims to discover the most amount of routes contained in the API. To do so, the mutation process will be executed only with mutation operators relative to modifying routes: `addRoute, removeRoute` and `modifyRoute`.

Route Name	VR / GR	FP / DP	FP / DP Percentage
`/album`	114/114	19/30	63.33%
`/artist`	102/102	12/15	80%
`/track`	118/118	11/28	39.28%

Table 10: Results obtained for 20 iterations by *MutGPT* for 3 routes of the *Deezer* API, using only mutation operators relative to discovering parameters.

VR / GR	FR / DR	FR / DR Percentage	UFR
171/171	11/15	73.33%	2

Table 11: Results obtained for 20 iterations by *MutGPT* with the *Deezer* API, using only mutation operators relative to discovering routes.

Table 11 displays the existing routes found by *MutGPT*. As shown, the tool was able to discover 11/15 documented routes, resulting in a 73.33% route discovery percentage. Similarly to the previous experiment with parameters, the request validity rate is not exact as JSON data containing error messages was found.

By inspecting generated requests, *MutGPT* was able to correctly discover routes such as `/artist/{id}` with existing ID values of artists. The tool also found 2 undocumented routes, `/comments` and `/folder`, which proved to be false positives as they do not exist in the API.

Overview: Research Question 3

To conclude the third and last research question, *MutGPT* proved to correctly and sufficiently generate API specifications. For the 3 tested APIs, the average route discovery percentage sums up to 82.49%, considering the additional 15 routes of the *GBIF Species* API. The average parameter discovery percentage sums up to 75,10%[a].

[a]The average parameter discovery percentage was calculated with the parameter discovery rates of the *GBIF Species* API and the *Random User Generator* API, along with the parameter discovery rate of the 42/73 found parameters for the 3 tested routes of the *Deezer* API. To recall, this last percentage corresponds to 57.53%.

5.5 Threats to Validity

Comparably to other scientific research, the current work is subject to a series of internal and external validity threats. Such threats are described in the following sections.

5.5.1 Internal Validity

Regarding the internal validity, it is possible that *MutGPT* or program implementations used for the research question experiments contain faults. This issue could potentially lead to incorrectly presented results. In order to mitigate such threat, each experimentation was carefully analyzed in an in-depth manner. For instance, even though the testing program contained a functionality to automatically display the number of valid requests after

a mutation process, status codes and response data were manually inspected to verify the behavior of generated request.

Moreover, tables found in Section 5.3 and Section 5.4 display the number of undocumented routes and parameters found during the evaluations. While this indicator can in certain scenarios reveal genuine undocumented routes or parameters, it often results in invalid elements. For this reason, each research question clearly states the importance of the interpretation of such result. For instance, Table 7 presents a truly undocumented route of the *GBIF Species* API, as it was verified afterwards by the relevant developers. Additional detail about this feedback is presented in Section 6.2.

Similarly, the experiments for the *Deezer* API in Section 5.4.3 display that all generated requests were valid. However, such result simply states that the requests returned status codes in the 200 range, which can still conceal errors depending on the API. Section 4.3 explains the importance of not referring to status codes exclusively.

As only 3 routes of the *Deezer* API were evaluated, the average parameter inference rate of the API might be different depending on the parameter discovery rate of non-evaluated routes. Thus, such result should be interpreted accordingly and only illustrates the potential capabilities of *MutGPT* when analyzing a larger API.

To conclude the section, a last aspect to consider is the specified number of documented routes and parameters of tested APIs. This amount was thoroughly verified for each API on the relevant documentation websites, which is up-to-date as of June 2023.

5.5.2 External Validity

For the external validity, the main threat concerns the selection of APIs used to demonstrate research questions and more importantly the capabilities of the *MutGPT* tool. In order to minimise the external validity threat, 6 different APIs were used, varying in terms of popularity, usage and application domain. To represent such diversity, Table 12 displays each utilized API alongside its application domain and usage reference. As the leveraged Large Language Models potentially contains more data for popular APIs, less popular APIs such as the *Random User Generator* API were used to demonstrate that satisfactory results could be attained.

API Name	Application Domain	Information
GBIF Species API	Biology	Works with data found in the *GBIF Checklist Bank*, containing over 10 million records [43].
OpenWeatherMap API	Weather	Allows access to weather data in over 200,000 cities worldwide [42].
MusicBrainz API	Music	As of February 2023, contains data regarding 2 million artists and 28 million recordings [54].
REST Countries API	Countries	Previously used by the *Spotify International Pricing Index* [69].
Random User Generator API	User Generation	No Information.
Deezer API	Music	API of *Deezer*, one of the most popular music streaming service in the world, with over 9.4 million total subscribers as of June 2022 [27].

Table 12: Details regarding the utilized APIs for the evaluation process.

6 Discussion

As evaluations and results have demonstrated the use of *MutGPT* in practice, a discussion can be now be held in order to summarize the tool's effectiveness.

6.1 Research Question Summary

First of all, the impact of the model temperature - the randomness of model responses - was analyzed in Section 5.2. Results demonstrated that a temperature of 0.7, which is the default temperature utilized by *ChatGPT*, provided the best requests in terms of validity and diversity. A temperature of 0 was able to generate valid but less diversified requests, while a temperature of 2 did not generate valid requests at all. Thus, **the temperature parameter has an impact on requests generated by the model in terms of validity and diversity**. Thus, as the research question demonstrated that a temperature providing slight response randomness displayed optimal results, the temperature value of 0.7 was chosen as parameter for prompts to the Large Language Model.

Second, *MutGPT* was compared against other preliminary strategies in Section 5.3, as a second research question. Results demonstrated that the Basic Request Prompt Strategy and the Complex Request Prompt Strategy succeeded overall at generating valid and diversified requests for existing APIs, the complex strategy displaying preferable outcomes compared to the basic strategy. Then, the Single Request Mutation Strategy exhibited a high percentage of generated valid requests, however with slightly inferior request diversity compared to the CRPS. This reduction in the diversity of generated requests was supported by the fact that the SRMS relies on a single initial seed, upon which new mutations can be generated. In fact, while the CRPS strategy is efficient at generating a few requests, its performance diminishes the more amount of requests is required due to the model response limit and the model cost. For this reason, the Single Request Mutation Strategy was preferred, allowing further request generation. Expanding for this initial request mutation strategy, the Multiple Request Mutation Strategy utilized by *MutGPT* displayed superior request generation with finer optimization. As the initial mutation strategy accomplished a single mutation per model prompt, the improved strategy was able to perform multiple mutations with a single prompt by requesting all possible mask values directly. Consequently, the research question proved that ***MutGPT* is more performing than preliminary strategies in terms of request generation, validity and optimization**.

Third, as *MutGPT* proved to be the pre-eminent strategy for generating requests, its capabilities at automatically generating API specifications needed to be tested. To recall, automatically generating API specifications consisted in the base idea of this Master's Thesis. In order to do so, the third and last research question described in Section 5.4 consisted in comparing the specification inferred by the tool with the existing documentation of APIs. Observed results display that ***MutGPT* is capable of sufficiently inferring the specification of APIs, with an average route discovery rate of 82.49% and an average parameter discovery rate of 75.10% regarding the tested APIs**. Moreover, the tool was able to discover 2 undocumented routes of the *GBIF Species* API, which is promising regarding the initial research question of discovering API elements without documentations. Thus, ***MutGPT* is capable of discovering valid and undocumented routes for a tested API**. Even though false positives were observed - mostly regarding parameters and invalid requests flagged as valid, *MutGPT* generated in most cases diverse and truly valid requests.

To summarize, the research questions demonstrated that **(1) Large Language Models can generate valid and diversified HTTP requests for RESTful APIs** and that **(2) it is possible to automatically infer RESTful API specifications by leveraging Large Language Models**. Even though *MutGPT* contains certain issues such as the discovery of invalid routes or parameters due to false positives, future work can expand upon its foundation in order to build even more capable tools leveraging Large Language Models. However, the training data limit of models cannot be directly improved, as it depends on the relevant developers to release new or improved models containing up-to-date training data.

6.2 Undocumented Route Feedback

As detailed in Section 5.4.3, *MutGPT* was able to discover 2 undocumented routes of the *GBIF Species* API: `/species/lookup` and `/species/{id}/identifier`. Accordingly, an email was sent to the *GBIF* API help desk in order to verify if the routes are truly undocumented. Shortly after, a data analyst of the API responded to the email, stating that the routes are in fact undocumented on the main *GBIF* documentation website. According to the correspondent, the discovered routes are probably deprecated, but remain functional.

This feedback proves that not only is *MutGPT* capable of inferring documented API specifications, it is also capable of inferring undocumented API specifications based on the validity of the found specifications w.r.t. the API endpoint.

7 Future Work

Regarding future work, various expansions are especially possible as the leveraged Large Language Model technologies continue to grow and expand. For the work of this Master's Thesis, multiple improvements are described in the following paragraphs.

Testing Strategies with Newer Models. As newer and more performing models are released, a starting point would consist in testing the preliminary strategies on those newer models. For instance, as the new `gpt-4-32k` model contains an enormous token limit of 32,000 for responses and exhibits enhanced capabilities compared to the models used in this work, the practical use of the initial Direct Specification Prompt Strategy could be reconsidered. Indeed, directly generating a complete and formal *OpenAPI* specification of an API could be possible in a near future.

Generating *OpenAPI* Specifications. As the present work demonstrated that it is possible to sufficiently infer API specifications, a possible improvement would be to directly generate an *OpenAPI* specification as output. Such task would require additional parsing and structuring of generated outputs. However, it would enable the use of *MutGPT* outputs directly as inputs of various API testing tools, which require formal *OpenAPI* specifications.

Correcting Invalid Requests with Prompts. As modern models can be re-prompted while preserving previous conversational data, invalid requests could be sent back to the leveraged model in order to attempt to find fixes and improve future request generation. This strategy would allow the model to avoid previously incorrect generations, thus enhancing the request mutation process. Moreover, the leveraged Large Language Models could manage the request validity verification. By feeding request response data returned from the corresponding server endpoint, models could detect issues such as syntactically invalid requests or in-page errors. For GET HTTP requests, a model could even be capable of verifying if the returned response data by the server corresponds to the initial request, testing the behaviour of the API.

Dynamic Mutation Operator Choice. Another possible improvement of *MutGPT* consists in removing or reducing the randomness in the choice of request seeds and mutation operators. For instance, the results of the Multiple Request Mutation Strategy in Section 5.3.3 demonstrated that a lot of mutated requests are invalid due to mutation operators responsible for changing routes, resulting in a lower percentage of valid requests. A possible solution to such issue would consist in dynamically attributing a certain score to each mutation operator, reflecting the effectiveness of the mutation operator with the tested API. Thus, when mutating, the operators with higher scores would be prioritized over the operators with lower scores. To illustrate, if a certain API always results in invalid requests when adding a new route, this surely signifies that the API does not contain additional route depth. The `addRoute` operator would then have a low mutation score, which would result in it being chosen much less often than other potentially interesting mutation operators.

To conclude the current section, future work regarding the use of Large Language Models is achievable. Possibilities are seemingly endless, relying on the ever growing capabilities of such models.

8 Conclusion

Initially, an issue was brought up in this Master's Thesis: API documentations are often incomplete, informal or nonexistent. This issue hinders the understanding of developers willing to leverage APIs, and hinders the API testing tools which often require a formal specification as input.

To solve this issue, a solution was proposed: **Leveraging Large Language Models to Automatically Infer RESTful API Specifications**. This idea would take advantage of the prominent Large Language Model technologies, such as the models provided by *OpenAI*.

In order to leverage such technologies, initial research begun by testing *ChatGPT* and the *OpenAI Python* library. A couple of issues were uncovered, such as the cost of prompting models, the response time and the token limit. Preliminary findings consisted in the exploration of various strategies solving the initial documentation issue. A first strategy, the **Direct Specification Prompt Strategy**, attempted to directly prompt the model to generate API specifications. However, the token limit and *hallucinations* of the model secluded this strategy. The **Basic Request Prompt Strategy** and the **Complex Request Prompt Strategy** fixed such problems by prompting the model for API requests, which do not contain a lot of tokens and can be easily verified by sending each request to the corresponding API endpoint. The strategies proved to correctly generate valid and varied requests, the CRPS demonstrating superior results. Nonetheless, such strategies encountered similar difficulties when prompting the model for a large amount of requests. Thus, a new strategy was explored: **request mutation**.

The **Single Request Mutation Strategy** consisted in randomly masking - hiding - a route, a parameter or a value of a request, in order to generate small variations of such requests. Each masked request would then be sent to the leveraged Large Language Model, which was prompted to respond with a new request containing a valid replacement for the mask. Thus, if a new mutated request proved to be valid when sending it to the API endpoint, its content could be automatically inferred in the API specification. Such process was implemented in a tool entitled ***MutGPT***. While the tool proved to be effective, an issue persisted: Optimization. Indeed, each mutation required a request to the *OpenAI* endpoint.

To solve this issue, an improved **Multiple Request Mutation Strategy** was implemented in *MutGPT*. The solution would still execute the initial mask procedure, but would ask the model to give all possible values that could replace the mask in the given request. By doing so, a single prompt to the *OpenAI* server could result in multiple mutations at the same time.

For the purpose of formally evaluating the tool, **3 research questions** were formulated:

- *What is the impact of the prompt temperature parameter on the Large Language Model predictions, in terms of request validity and diversity?* **(RQ1)**
- *Is MutGPT more performing than the preliminary strategies in terms of generating valid and diverse requests?* **(RQ2)**
- *Can MutGPT infer the documentation of APIs, and possibly discover undocumented features?* **(RQ3)**

Each research question was then analyzed with multiple experiments on 5 different APIs: the *GBIF Species* API, the *OpenWeatherMap* API, the *MusicBrainz* API, the *REST Countries* API, the *Random User Generator* API and the *Deezer* API.

First, results of RQ1 proved that **the temperature parameter has an impact on requests generated by the model in terms of validity and diversity**, and that specifying a temperature with slight randomness displayed the best results.

Second, results of RQ2 proved that ***MutGPT* is more performing than preliminary strategies in terms of request generation, validity and optimization**.

Third, results of RQ3 proved that ***MutGPT* is capable of sufficiently inferring the specification of APIs** and that ***MutGPT* is capable of discovering valid and undocumented routes for a tested API**.

Then, possible improvements of the work were detailed, regarding the future capabilities of Large Language Models and the possible upgrades of *MutGPT*.

Overall, this Master's Thesis uncovered 2 new contributions:

1. Large Language Models are capable of generating valid and diversified HTTP requests for RESTful APIs, only requiring the name of the API as input.

2. It is possible to automatically infer RESTful API specifications by leveraging Large Language Models.

A replication package with the implementation and evaluation data is available at the following *GitHub* repository URL: `https://github.com/alixdecr/MutGPT`**.**

Acronyms

API Application Programming Interface

BERT Bidirectional Encoder Representations from Transformers

BRPS Basic Request Prompt Strategy

CRPS Complex Request Prompt Strategy

DSPS Direct Specification Prompt Strategy

GBIF Global Biodiversity Information Facility

GPT Generative Pre-trained Transformer

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

LLM Large Language Model

MAB Multi-Armed Bandit

MRMS Multiple Request Mutation Strategy

NLP Natural Language Processing

OAS OpenAPI Specification

REST Representational State Transfer

SRMS Single Request Mutation Strategy

URI Uniform Resource Identifier

URL Uniform Resource Locator

W.R.T. With Reference To

Bibliography

[1] 2015. Gias Uddin and Martin P. Robillard. How API Documentation Fails. In *Institute of Electrical and Electronics Engineers*, Volume 32, Issue 4. Pages 68-75.

[2] 2019. Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Volume 1 (Long and Short Papers). Pages 4171–4186.

[3] 2020. Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. Pages 725-736.

[4] 2020. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings (EMNLP)*. Volume EMNLP 2020. Pages 1536–1547.

[5] 2021. Alberto Martin-Lopez, Sergio Segura, Antonio Ruiz-Cortés. RESTest: automated black-box testing of RESTful web APIs. In *ISSTA 2021: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Pages 682-685.

[6] 2022. Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis and Yves Le Traon. CodeBERT-nt: code naturalness via CodeBERT. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*.

[7] 2022. Alberto Martin-Lopez, Sergio Segura and Antonio Ruiz-Cortés. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Pages 408-420.

[8] 2023. Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fehmideh, Mst Shamima Aktar and Tommi Mikkonen. Towards Human-Bot Collaborative Software Architecting with ChatGPT. *arXiv preprint arXiv:2302.14600*.

[9] 2023. Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis and Yves Le Traon. Efficient Mutation Testing via Pre-Trained Language Models. *arXiv preprint arXiv:2301.03543*.

[10] 2023. Max Schäfer, Sarah Nadi, Aryaz Eghbali, Frank Tip. Adaptive Test Generation Using a Large Language Model. *arXiv preprint arXiv:2302.06527*.

[11] 2023. Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang and Lingming Zhang. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*.

[12] 2023. Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang and Lingming Zhang. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *arXiv preprint arXiv:2304.02014*.

[13] 2022. Gilles Perrouin. Boundary & Partition Testing [Lecture Slides]. University of Namur. `https://webcampus.unamur.be/login`. Pages 37-77.

[14] 2022. Gilles Perrouin. Fuzzing [Lecture Slides]. University of Namur. `https://webcampus.unamur.be/login`. Pages 1-12.

[15] 2022. Gilles Perrouin. Testing Fundamentals [Lecture Slides]. University of Namur. https://webcampus.unamur.be/login. Page 17.

[16] 2022. Xavier Devroey. Lexical Fuzzing [Lecture Slides]. University of Namur. https://webcampus.unamur.be/login. Pages 25-50.

[17] 2017. James Messinger. API Testing Tips from a Postman Professional. *Postman*. https://blog.postman.com/api-testing-tips-from-a-postman-professional. Last accessed 2023-05-28.

[18] 2017. Mark Amery. OK to skip slash before query string? *Stack Overflow*. https://stackoverflow.com/questions/1617058/ok-to-skip-slash-before-query-string. Last accessed 2023-05-17.

[19] 2018. Tomer Bar. Notifying our Developer Ecosystem about a Photo API Bug. *Meta*. https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug. Last accessed 2023-05-06.

[20] 2020. Alberto Martín López. RESTest: Automated Black-Box Testing of RESTful Web APIs. *YouTube*. https://www.youtube.com/watch?v=TnGkwMDBDt4. Last accessed 2023-05-28.

[21] 2020. What is a REST API? *Red Hat*. https://www.redhat.com/en/topics/api/what-is-a-rest-api. Last accessed 2023-06-06.

[22] 2021. Alberto Martín López. RESTest: Your RESTful APIs. Tested. By robots. *YouTube*. https://www.youtube.com/watch?v=-rydj3T_YjA. Last accessed 2023-05-28.

[23] 2021. Darshana. 'https://restcountries.eu/rest/v2/all' not working, How to get all countries now? *Stack Overflow*. https://stackoverflow.com/questions/69363029. Last accessed 2023-05-30.

[24] 2021. HTTP responses. *IBM*. https://www.ibm.com/docs/en/ctsfz/5.2?topic=protocol-http-responses. Last accessed 2023-06-06.

[25] 2022. Abhinav Asthana. Celebrating 20 Million Postman Users. *Postman*. https://blog.postman.com/celebrating-20-million-postman-users. Last accessed 2023-05-28.

[26] 2022. Lokesh Gupta. What is REST. *REST API Tutorial*. https://restfulapi.net/. Last accessed 2023-06-06.

[27] 2022. Deezer reports improved profitability and double-digit revenue growth in H1 2022. *Deezer Investors*. https://www.deezer-investors.com/wp-content/uploads/2022/09/EN-Deezer_H1-2022_Results_Press-Release-2022-09-21-FINAL.pdf. Last accessed 2023-06-02.

[28] 2023. Adrian Tam. What are Large Language Models. *Machine Learning Mastery*. https://machinelearningmastery.com/what-are-large-language-models. Last accessed 2023-06-06.

[29] 2023. Haziqa Sajid. What Are LLM Hallucinations? Causes, Ethical Concern, & Prevention. *Unite.AI*. https://www.unite.ai/what-are-llm-hallucinations-causes-ethical-concern-prevention. Last accessed 2023-06-06.

[30] 2023. Mike Igartúa. ChatGPT is learning, but still lacks real life experience! *Reddit*. https://www.reddit.com/r/Funnymemes/comments/10ohd2n/chatgpt_is_learning_but_still_lacks_real_life. Last accessed 2023-05-15.

[31] 2023. HTTP requests. *IBM*. https://www.ibm.com/docs/en/ctsfz/5.3?topic=protocol-http-requests. Last accessed 2023-06-06.

[32] 2023. Number of monthly active Facebook users worldwide as of 1st quarter 2023. *Statista*. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide. Last accessed 2023-05-06.

[33] 2023. Rapid's New State of APIs Report Finds Surge in Usage Leading into 2023 with More Companies Planning to Monetize APIs. *Business Wire*. https://www.businesswire.com/news/home/20230111005140/en. Last accessed 2023-05-06.

[34] [n.d.]. Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser and Christian Holler. Fuzzing: Breaking Things with Random Inputs. *The Fuzzing Book*. https://www.fuzzingbook.org/html/Fuzzer.html. Last accessed 2023-06-06.

[35] [n.d.]. Manraj and Sudhakar Kumar. Generative Pre-trained Transformer. *Insights2Techinfo*. https://insights2techinfo.com/generative-pre-trained-transformer. Last accessed 2023-06-06.

[36] [n.d.]. Allure Framework. *GitHub*. https://github.com/allure-framework. Last accessed 2023-05-28.

[37] [n.d.]. API. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/API. Last accessed 2023-05-04.

[38] [n.d.]. Black box testing. *Javatpoint*. https://www.javatpoint.com/black-box-testing. Last accessed 2023-06-06.

[39] [n.d.]. ChatGPT. *OpenAI*. https://chat.openai.com. Last accessed 2023-05-03.

[40] [n.d.]. ChatGPT. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/ChatGPT. Last accessed 2023-05-11.

[41] [n.d.]. Chuck Norris Jokes API. *Chucknorris*. https://api.chucknorris.io. Last accessed 2023-05-24.

[42] [n.d.]. Current weather data. *OpenWeatherMap*. https://openweathermap.org/current. Last accessed 2023-06-02.

[43] [n.d.]. Dataset search. *GBIF*. https://www.gbif.org/dataset/search?type=CHECKLIST. Last accessed 2023-06-02.

[44] [n.d.]. Deezer API. *Deezer for developers*. https://developers.deezer.com/api. Last accessed 2023-06-02.

[45] [n.d.]. Fuzzing. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Fuzzing. Last accessed 2023-06-06.

[46] [n.d.]. Genetic Algorithms - Mutation. *Tutorialspoint*. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation. Last accessed 2023-06-06.

[47] [n.d.]. Helioviewer API. *The Helioviewer Project*. https://api.helioviewer.org/docs/v2. Last accessed 2023-05-24.

[48] [n.d.]. HTTP. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/HTTP. Last accessed 2023-06-06.

[49] [n.d.]. HTTP - Requests. *Tutorialspoint*. https://www.tutorialspoint.com/http/http_requests. Last accessed 2023-06-06.

[50] [n.d.]. Language Model. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Language_model. Last accessed 2023-06-06.

[51] [n.d.]. Large Language Model. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Large_language_model. Last accessed 2023-06-06.

[52] [n.d.]. Models. *OpenAI*. https://platform.openai.com/docs/models. Last accessed 2023-05-15.

[53] [n.d.]. Multi-armed bandit. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Multi-armed_bandit. Last accessed 2023-06-06.

[54] [n.d.]. MusicBrainz. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/MusicBrainz. Last accessed 2023-06-02.

[55] [n.d.]. MusicBrainz API. *MusicBrainz*. https://musicbrainz.org/doc/MusicBrainz_API. Last accessed 2023-06-08.

[56] [n.d.]. MusicBrainz API / Rate Limiting. *MusicBrainz*. https://musicbrainz.org/doc/MusicBrainz_API/Rate_Limiting. Last accessed 2023-05-29.

[57] [n.d.]. Natural Language Processing. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Natural_language_processing. Last accessed 2023-06-06.

[58] [n.d.]. Neural Network. *Togaware*. https://datamining.togaware.com/survivor/Neural_Network.html. Last accessed 2023-05-28.

[59] [n.d.]. *OpenAI*. https://openai.com. Last accessed 2023-05-03.

[60] [n.d.]. OpenAI Codex. *Lablab*. https://lablab.ai/tech/openai/codex. Last accessed 2023-05-11.

[61] [n.d.]. OpenAPI Specification. *Swagger*. https://swagger.io/specification. Last accessed 2023-06-06.

[62] [n.d.]. OpenAPI Specification. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/OpenAPI_Specification. Last accessed 2023-06-06.

[63] [n.d.]. OpenWeatherMap API. *OpenWeatherMap*. https://openweathermap.org/api. Last accessed 2023-06-06.

[64] [n.d.]. Playground. *OpenAI*. https://platform.openai.com/playground. Last accessed 2023-05-17.

[65] [n.d.]. *Postman*. https://www.postman.com. Last accessed 2023-05-07.

[66] [n.d.]. Random User Generator API. *Randomuser*. https://randomuser.me. Last accessed 2023-06-01.

[67] [n.d.]. Representational state transfer. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Representational_state_transfer. Last accessed 2023-06-06.

[68] [n.d.]. *REST Assured*. https://rest-assured.io. Last accessed 2023-05-07.

[69] [n.d.]. REST Countries. *GitHub*. https://github.com/apilayer/restcountries. Last accessed 2023-06-02.

[70] [n.d.]. REST Countries API. *REST Countries*. https://restcountries.com/. Last accessed 2023-06-08.

[71] [n.d.]. RESTest. *GitHub*. https://github.com/isa-group/RESTest. Last accessed 2023-05-28.

[72] [n.d.]. RESTler. *GitHub*. https://github.com/microsoft/restler-fuzzer. Last accessed 2023-05-07.

[73] [n.d.]. Species API. *GBIF*. https://www.gbif.org/developer/species. Last accessed 2023-05-24.

[74] [n.d.]. TestPilot. *GitHub Next*. https://githubnext.com/projects/testpilot. Last accessed 2023-05-28.

[75] [n.d.]. The 7 Software "-ilities" You Need To Know. *Codesqueeze*. http://codesqueeze.com/the-7-software-ilities-you-need-to-know. Last accessed 2023-05-06.

[76] [n.d.]. Tokenizer. *OpenAI*. https://platform.openai.com/tokenizer. Last accessed 2023-05-15.

[77] [n.d.]. Web API. *Spotify for Developers*. https://developer.spotify.com/documentation/web-api. Last accessed 2023-05-24.

[78] [n.d.]. What Is An API (Application Programming Interface)? *Amazon Web Services*. https://aws.amazon.com/what-is/api. Last accessed 2023-05-04.

[79] [n.d.]. What Is A Neural Network? *Amazon Web Services*. https://aws.amazon.com/what-is/neural-network. Last accessed 2023-06-06.

[80] [n.d.]. What Is A RESTful API? *Amazon Web Services*. https://aws.amazon.com/what-is/restful-api. Last accessed 2023-06-06.

[81] [n.d.]. What is natural language processing (NLP)? *IBM*. https://www.ibm.com/topics/natural-language-processing. Last accessed 2023-06-06.