Multi-Cloud API Approach to Prevent Vendor Lock-In

Department of Computer Science and Engineering
COEN 241: Cloud Computing
Professor Ming-Hwa Wang
Santa Clara University
Winter Quarter 2021

Angeline Chen
Jayavardhani Kathika
Kent Ngo

ACKNOWLEDGEMENT

We express our deep sense of gratitude to Professor Ming-Hwa-Wang for teaching us the fundamentals of Cloud Computing and encouraging us to complete this project.

Table of Contents

List of Tables

List of Figures

Abstract

Vendor lock-in is a situation where a customer gets struck with a vendor and is unable to use services offered by another vendor. This is also known as proprietary lock-in. Also customers have to pay huge switching costs if they plan to move to another vendor, so to overcome those charges many customers continue with a single vendor even though their services are no longer efficient. In cloud computing vendor lock-in is caused due to the platform dependent services that the cloud providers offer. One solution to overcome this problem is to standardize the cloud APIs. By standardizing the APIs the user can access multiple APIs from different cloud providers through one standardized API call which prevents vendor lock-in. The main scope of this project is to design and implement a web server which provides results for various machine learning APIs by getting results from multiple clouds which offer that API.

Keywords: Vendor lock-in, Cloud Computing, Standardizing APIs.

1. Introduction

1.1 Objective

The objective is to avoid the vendor lock-in effect in the deployment of applications to the cloud by standardizing cloud API calls using wrappers.

1.2 Problem

Many organizations have developed their software using a single cloud service provider, such AWS and Azure and face problems, such as interoperability and portability issues of cloud computing due to the lack of integration between cloud service providers. There are no widely accepted standards and each cloud service provider offers their own unique proprietary services and technologies which are non-compatible with other cloud providers. Therefore, customers are not able to easily port their applications or switch to vendors easily. Due to the incompatibility issues, porting an application to another vendor would cause the customer to re-engineer their entire application to fit their platform. As a result, customers tend to stay with their current cloud service provider because they don't want to pay a lot of time and money to port their applications.

However, the problem with a single cloud service provider is that customers cannot take advantage of using the best quality of resources and services from other cloud providers and end up becoming "vendor locked-in". In a vendor lock-in situation, customers have become too dependent on using their single cloud service provider to host their application that if the cloud service provider's prices were to dramatically increase, the customer can't do anything because

porting their application would result in starting all over. The longer a customer stays with their cloud service provider, the more inclined the customer would want to stay because their entire infrastructure was built on it. As a result, customers are put into risks of slowly losing a lot of money if they're stuck to using an inferior product. Additionally, if a single cloud service provider were to go out of business or their database servers were to crash, this would lead to a substantial negative impact on the customers and cause them to lose everything. Overall, from these consequences of vendor lock-in, dominant cloud service providers can take advantage of the situation and use these tactics to stifle competition and deprive customers of better prices.

1.3 Why is this Project related to this Class?

This project is related to this class because it tackles the business perspective issues of cloud computing. Cloud computing is renowned for its on-the-demand services, and "pay what you use" motto. Most companies want to save as much money as possible and with the right circumstances, cloud computing can become very beneficial for start up companies, who are not as experienced with IT or who want to take advantage of their microservices. However, vendor lock-in can be a problematic obstacle for these companies as it prevents them from saving money and achieving the best prices for the microservices. With vendor lock-in, customers are always at risk of paying more than usual.

1.4 Why other approaches are not good

Although most papers can describe their Multi-Cloud architecture, they cannot demonstrate how much of a positive impact it may create due to the lack of data comparisons. They do not consider how efficient their methodology is, but rather focus on how flexible their

architecture is by using multiple tools and resources. It can create excessive amounts of complexity due to their considerable amount of high-level layers.

1.5 Why you think your approach is better

Our approach abstracts all the APIs of different clouds. The customers don't have to limit themselves to a single cloud, they can use various services from different clouds. They don't have to think about the structure of APIs of different clouds; they can just access the services from different clouds with a single API provided by us for a particular service and get the relevant service from the cloud that provides it.

1.6 Scope of Investigation

In our scope of investigation, we would be looking into these terms and see how they would help us determine how to prevent vendor lock-in:

- Multi-cloud approach: Using multiple cloud vendors to host the customer’s infrastructure
- Portability: Testing how easy it is to migrate an application
- Interoperability: Making sure the software works on various cloud platforms
- Containerization: Simplifies software deployment on various platforms

With these strategies, we would be able to create a solution to provide web services, which would operate independent from proprietary cloud service provider’s technologies while implementing modern standards and technologies.

2. Theoretical Bases and Literature Review

2.1 Definition of the Problem

The major issue of vendor lock-in is that customers are unable to port their applications or unable to switch to another cloud vendor because of the lack of integration and interoperability among cloud service providers. Unlike Internet service providers where customers can switch easily to another provider when they're not getting quality of service, cloud service providers lock their customers in even when they're not getting the best quality service or the provider is beginning to decline. Large cloud vendors know that it's very difficult to port their applications and know that they can monopolize their resources. So the vendors have proposed their own solutions with proprietary formats and interfaces instead of following widely accepted standards. As a result, they control the market and prevent customers from getting the best services. Most common causes of vendor lock-in are unique file formats, incompatible or proprietary APIs, and lack of modern standardizations.

2.2 Related Research to Solve the Problem

2.2.1 Identifying Adaptation Needs to Avoid the Vendor Lock-in Effect in the Deployment of Cloud SBAs [1]

2.2.1.1 Adaptation Cases

The paper describes the modifications of a component detached from the application and deployed in a cloud environment in three scenarios: component-to-component, component-to-cloud, and component-to-SaaS.

In the component-to-component scenario, adaption between components modifies communication interfaces because the communication interfaces between the component and the remaining components in a cloud environment and a non-cloud environment are different. The sources of mismatch are service naming and location, different technologies, and operation ordering mismatch because the name and the location of services and operations are different, the technologies used by components are different, and the order of operation invocations is different.

In the component-to-cloud scenario, adaption between components and cloud specific services modifies service dependencies because platform specific service dependencies in a non-cloud environment and cloud specific service dependencies in a cloud environment are different. The sources of mismatch are service naming, different technologies, operation ordering mismatch, SLA related requirements, and cloud-specific performance services because the name of services and functions are different, the APIs are different, the order of operation invocations is different, the SLAs are different, and the service composition of components is different.

In the component-to-SaaS scenario, adaption using third-party components or services (SaaS) modifies external services because the use of external services in a cloud environment and a non-cloud environment is different. The sources of mismatch are semantic and context-aware functionalities because the use of external services is different.

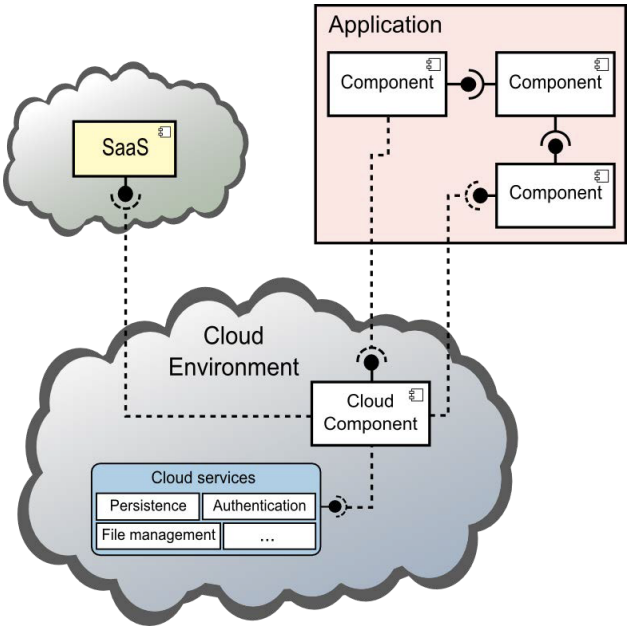

Figure 1 [1]: Scenario without Adaptors

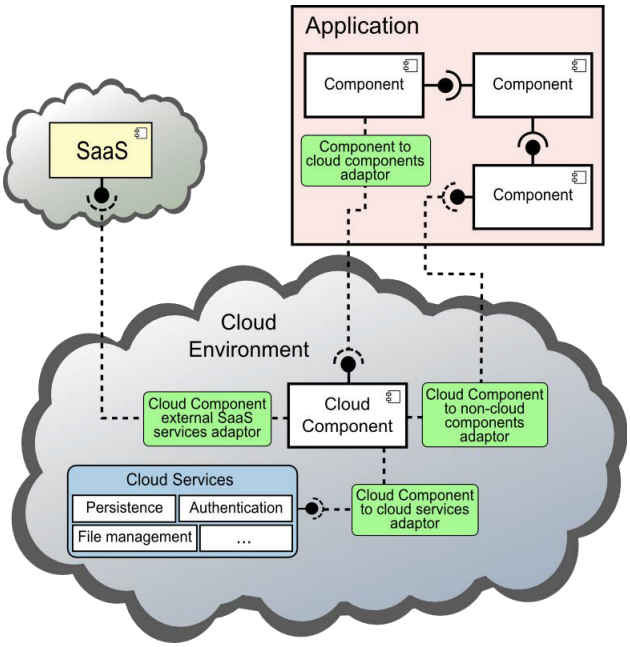

Figure 2 [1]: Scenario with Adaptors

2.2.1.2 Interoperability Levels

The paper also describes mismatches at four interoperability levels: signature, behavior, service, and semantic.

Signature adaptation deals with interoperability issues and differences in service names, parameters, data representation, encoding standards, and operation calling conventions. Wrappers or stubs act as proxies between client components and service providers. Adaptors translate service names, cast, reorder, and synthesize parameters, and accommodate calling conventions and data representation mismatch. Adaptors allow the client to access a component that moved from a non-cloud environment to a cloud environment. Mappings establish one-to-one correspondences. Adaptors translate and transmit interactions.

Behavioral adaptation deals with protocol issues and differences in the order and granularity of interaction messages, which are service or operation requests and responses. Mappings establish correspondences between messages. Adaptors allow a service request issued by a client component to correspond to several invocations of the provider's interface.

Service adaptation deals with non-functional issues, such as security, cost, performance, and dependability and differences in QoS policies. Adaptors allow devices with different hardware and software resources and capabilities to negotiate SLAs.

Semantic adaptation deals with conceptual issues and differences in semantic descriptions. An ontology is a machine-readable representation that captures the semantic description of services and the relationships between concepts. Adaptors use ontologies to match semantics.

Interoperability Levels	Sources of Mismatch	Involved Scenarios
signature	service naming different technologies	component-to-component component-to-cloud component-to-SaaS
behavior	operation ordering mismatch	
service	SLA related requirements cloud-specific performance services	component-to-cloud component-to-SaaS
semantic	semantic and context-aware functionalities	component-to-SaaS

Table 1 [1]: Adaptation Cases and Interoperability Levels

2.2.2 Preventing Vendor lock-ins via an interoperable multi-cloud deployment approach [2]

2.2.2.1 Main idea

Vendor lock-in prevents the customers from adopting cloud services offered by various cloud service providers. One solution for preventing vendor lock-in is to adapt multi cloud strategy. Multi cloud enables the customers to interact with two or more cloud providers and work with them simultaneously. The problems with multi cloud are portability and interoperability.

Portability is the ability to move applications among multiple cloud platforms without changing the code. But the software stack and the features the existing clouds provide are different from one another. These differences prevent portability and promote vendor lock-in. Interoperability among cloud providers helps in multiple cloud deployments.

2.2.2.2 Single Cloud Site Architecture

A single three tier web service consists of presentation tier, application tier and data tier. In this case the application runs only on a single cloud service provider. This does not support portability and interoperability also if the cloud provider is down the whole application stops running. This architecture creates dependency on a single cloud provider. Avoiding dependency on a single cloud provider

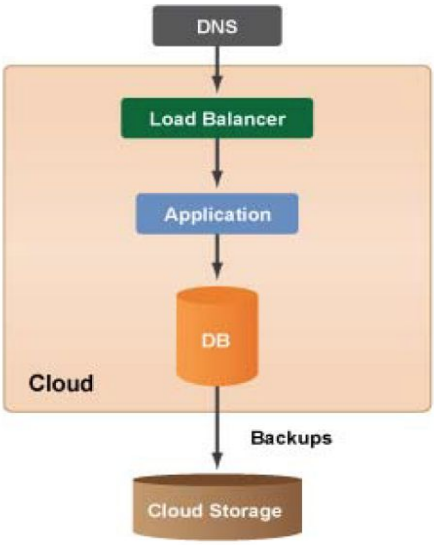

Figure 3 [2]: Single Cloud Site Architectures - Non-Redundant 3-Tier Architecture

2.2.2.3 Multi Cloud Architecture

Interoperability is achieved by adding an abstraction layer which is represented through a container orchestration and microservice approach. In micro service architecture the whole application can be divided into various services. These services can be deployed on various platforms. The services which are connected to each other interact via APIs. Docker supports microservices for deploying on Kubernetes.

To support replication, synchronization and scalability across various cloud service providers a NoSQL database is used. MongoDB supports replica sets and sharding. MongoDB instances are deployed under kubernetes.

Container orchestration helps to decouple platform specific APIs. This enables the customer to interact with various cloud providers using APIs of container orchestration environment. For the container orchestration Kubernetes is used. Kubernetes is an open source software and it can deploy and manage container engines like Docker.

An application modelling tool helps in building and scaling the container orchestration environment among different clouds. Juju charms along with Conjure-up are used in deploying kubernetes and also connecting with different APIs of cloud providers.

Therefore Kubernetes cluster is formed with kubernetes installed on three cloud providers. These three networks are connected via global network connection. A stateless load balancing mechanism which can be implemented via load balancer or DNS round robin will distribute the user requests among multiple cloud platforms and the data is redirected to the database.

In this way portability and interoperability is promoted and vendor lock-in is reduced.

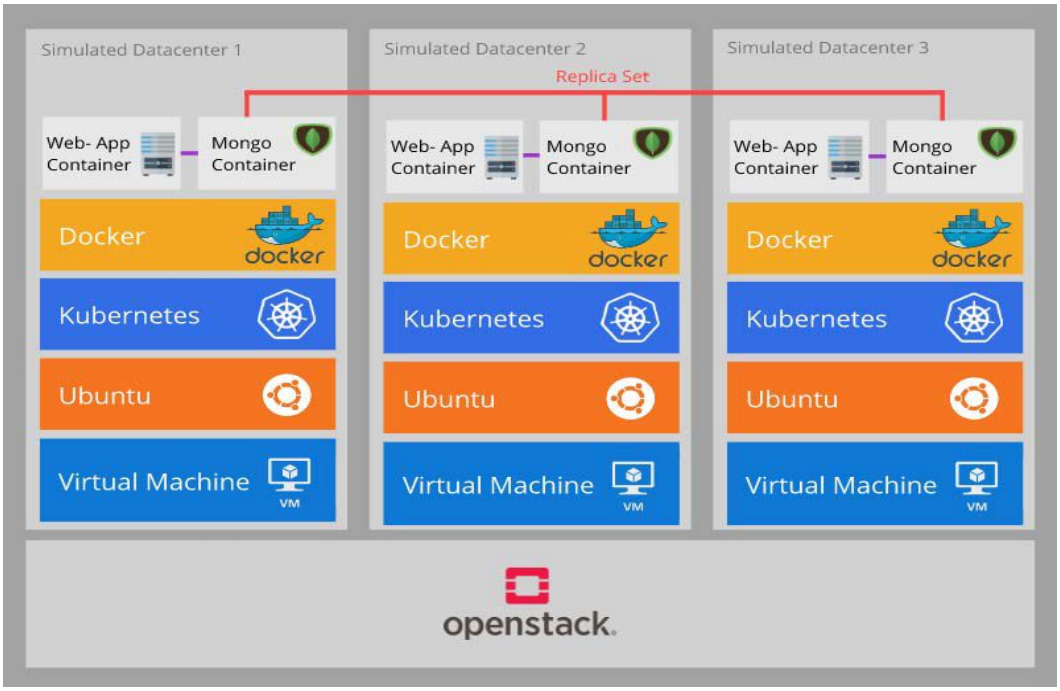

Figure 4 [2]: Prototype architecture

2.2.3 Multi-Cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-in [3]

2.2.3.1 Fixing Issues for Cloud Standardization

Although there are current solutions to support portability between cloud vendors, such as platform intermediation, the paper describes that they wouldn't work for every cloud vendor. Creating a common model transformation package from various cloud vendors is considered to be very difficult and time consuming. As a result, the paper tries to address this issue by creating a multi-cloud PaaS architecture that would prevent vendor lock-in.

The paper proposes to try to create a heterogenous platform that would work with different multi-cloud vendors. By analyzing the various PaaS platforms and libraries, they would be able to pick and choose the best resources that would support heterogenous development.

2.2.3.2 Tools and Architectural Design

To orchestrate the deployment, it uses P-TOSCA, which stands for Portable Topology and Orchestration Specification for Cloud Applications. It would offer standard app synthesis and design to PaaS platforms and would help resolve vendor lock-in situations due to its flexibility and . Additionally, it's an Open-Source API and it's XML based app topology.

Another tool used by the architecture is mOSAIC, which allows for monitoring and deploying dynamic reconfiguration. It would enhance orchestration and allow the clients to deploy fast and efficient changes to multiple clouds by using a unified API. Clients would not need to stop their app services, which could save them money.

Lastly, the architecture uses the Opscode-Chef as the configuration management tool. It fixes the issue managing applications from different PaaS platforms by offering many functionalities. It'll help create standards among their multiple clouds and give policies of how the cloud should be configured.

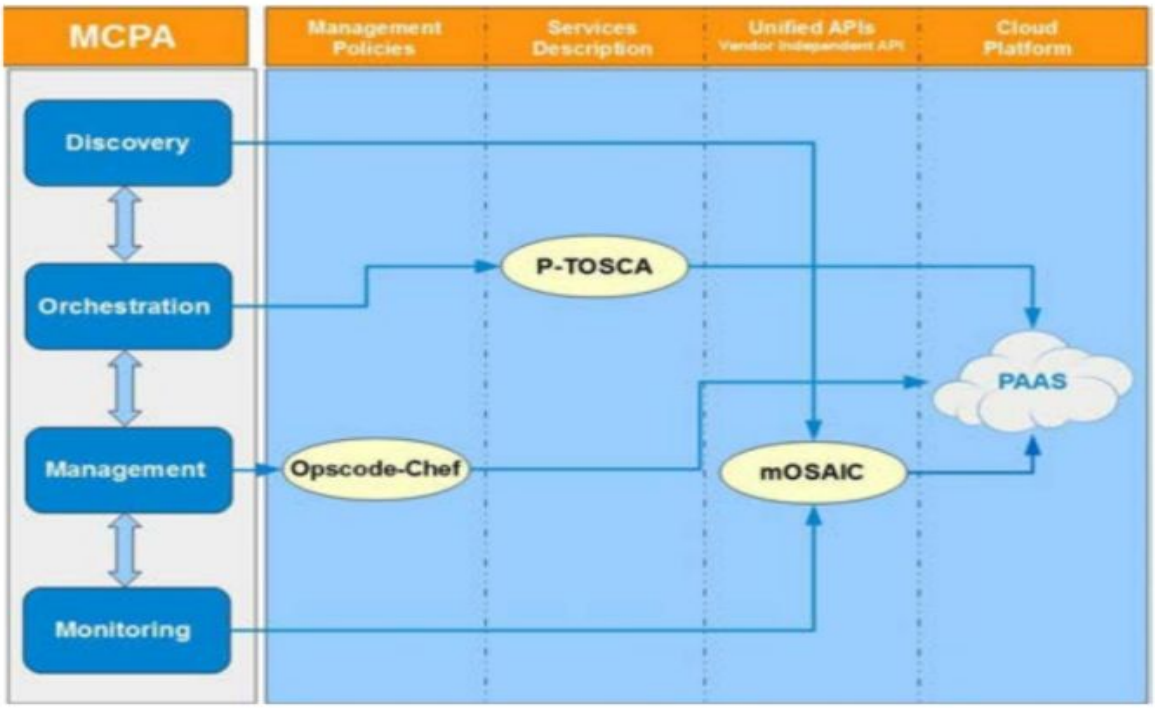

Figure 5 [3]. Initial High Level Architecture for MCPA

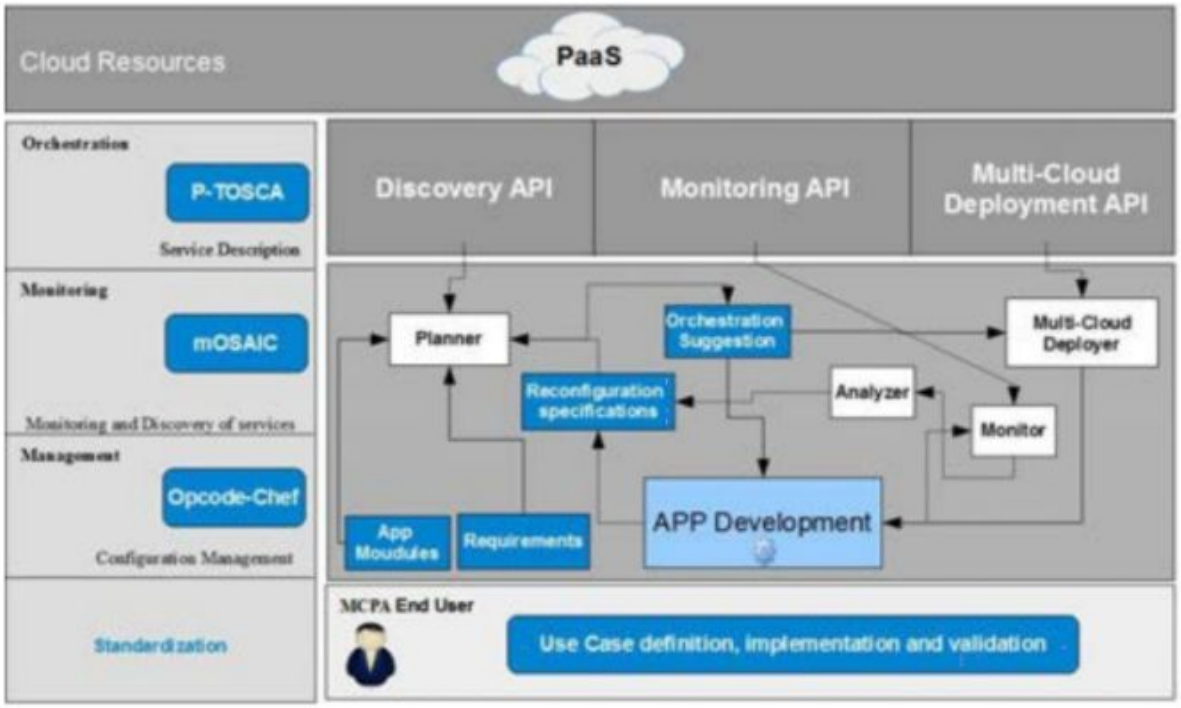

Figure 6 [3]: MCPA Modules, Relations and Entities

2.3 Advantages of the Research Paper

2.3.1 Identifying Adaptation Needs to Avoid the Vendor Lock-in Effect in the Deployment of Cloud SBAs

- **Advantages:**
    - The paper describes problems and solutions, which helped us identify the need for the standardization of cloud APIs.
    - The paper includes figures that compare scenarios with and without adaptors.

2.3.2 Preventing Vendor lock-ins via an interoperable multi-cloud deployment approach

- **Advantages:**
    - Promotes interoperability and portability between different clouds by abstracting the software layer which reduces the vendor lock-in.
    - This prototype does not require any additional testing as the core components are supported by major cloud service providers.

2.3.3 Multi-Cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In

- **Advantages:**
    - The tools they use such as P-TOSCA, mOSAIC, and Opscode-Chef are all open-source APIs that can be accessed by multiple cloud
    - Uses an unified API to synchronize deployment and monitoring.
    - Will incorporate standardization and administration services during application development
    - Compatible with other existing technologies and tools as well

2.4 Disadvantages of the Research Paper

2.4.1 Identifying Adaptation Needs to Avoid the Vendor Lock-in Effect in the Deployment of Cloud SBAs

- **Disadvantages:**
    - The paper does not describe implementation, which did not help us implement wrappers.
    - The paper does not include data.

2.4.2 Preventing Vendor lock-ins via an interoperable multi-cloud deployment approach

- **Disadvantages:**
    - Design is complex and this reduces productivity.

2.4.3 Multi-Cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In

- **Disadvantages:**
    - Does not work with all cloud platforms
    - There are no experiments to demonstrate the portability and elasticity of the architecture

2.5 Solution to the Problem

Because different cloud providers have their own platform specific APIs for various tasks, it is difficult to use a common API to a specific task on multiple cloud platforms. Not all the cloud providers support all the services. As a result, users need to use various clouds for various services, which results in a need to use different APIs on different cloud platforms.

Our solution is to standardize the APIs by getting the suitable API from a specific cloud provider according to the service the customer requests. Because there are various services a

cloud can offer, there is a need to standardize various services. For example, machine learning APIs provided by different cloud providers offer different services. While some services are offered by all cloud providers, other services are offered by only one cloud provider. Amazon, Microsoft, Google, and IBM provide speech and text processing APIs, image analysis APIs, and video analysis APIs. Users often use the cloud provider that offers the most services, but users sometimes use a different cloud provider that offers a unique service.

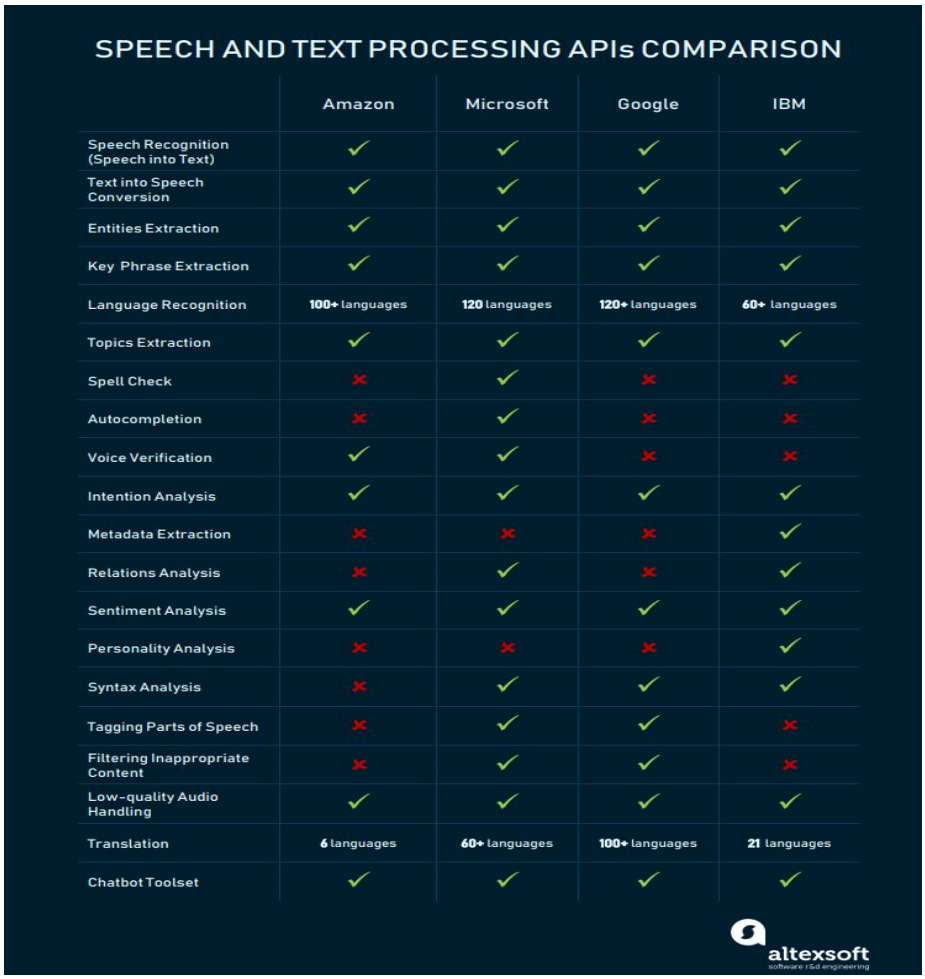

Figure 7 [4]: Microsoft's speech and text processing APIs have the most features, but Google's speech and text processing APIs support the most languages.

Figure 8 [4]: Google's image analysis APIs have the most features, but Amazon's and Microsoft's image analysis APIs support face recognition.

Figure 9 [4]: Microsoft's video analysis APIs have the most features, but Amazon's video analysis APIs support streaming videos.

Because of standardization and interoperability issues, it is difficult to switch between cloud providers. Cloud computing services offered by different cloud providers have different specifications. Proprietary technologies, such as cloud APIs, are incompatible. As a result, cloud API calls need to be standardized using wrappers.

2.6 Where your Solution Different from others

Our solution is different from other solutions because we will standardize cloud API calls instead of standardizing the deployment of applications to the cloud. We will standardize APIs of various services so that we can get our required service from different cloud providers with a single API and reduce vendor lock-in. A solution proposed by a paper is to create an abstraction layer through container orchestration solution as well as microservice approach which supports interoperability and portability in multi cloud. Another solution proposed by a paper that uses Open-Source tools, such as Opscode-Chef, P-TOSCA, and mOSAIC, could be used as the standard API.

2.7 Why your Solution is Better

Our solution is better because it is helping customers to use different services from different cloud providers rather than confining to a single cloud service. Also the customers don't have to think about different APIs of different clouds rather with a single API they can access the services. Our solution abstracts all the APIs of different cloud providers.

3. Hypothesis and Goals

3.1 Positive / Negative Hypothesis

If cloud API calls are standardized using wrappers, the number of lines in JSON responses returned by API calls will be reduced, and the round trip time of API calls will be reduced.

4. Methodology

4.1 Generating Input Data

To generate the output for this project, we would need to collect input text that we would like to analyze, such as key words and sentient. Therefore, we would be using Google Cloud and Microsoft Azure's API to analyze text processing and we believed that collecting reviews, such as restaurant reviews or hotel reviews from Yelp, would give us various and reliable input data to be analyzed. A sample response would be:

"Villa Brazil is such a wonderful little place! It is so clean it's amazing! Super comfortable everything! Kitchenette w fridge, microwave and coffee maker and all the supplies. I was here for the second time, but this time during the pandemic. I would go NOWHERE else during this time. I knew from my first visit how impeccably clean they are, so I wasn't worried. I'm sure they added extra precautions, it smelled so clean and taken care of. But it did before too. Fatima is a total gem! Her husband too! They go out of their way for your comfort and safety. Love them, their Villa and all that they do! Oh! Their cafe with home cooked amazing meals is the icing on the cake. The food is wonderful! And well worth every penny! Delivered right to your room! Thank you Fatima and team!"

4.2 How to Solve the Problem

Our approach is to write code that makes API calls compatible with different cloud providers, using machine learning APIs as an example. We will create a web server which makes the API calls and use JSON responses returned by API calls to test our wrappers. We will

measure the number of lines in JSON responses returned by API calls and the round trip time of API calls. If our hypothesis is correct, the number of lines in JSON responses returned by API calls is less when users call our wrappers than when users call APIs provided by different cloud providers, and the round trip time of API calls is less when users call our wrappers than when users call APIs provided by different cloud providers.

4.2.1 Algorithm Design

We will use the following pseudocode to implement wrappers that call APIs provided by different cloud providers:

```
if Microsoft's API has the feature
    call Microsoft's API
if Google's API has the feature
    call Google's API
combine the results of the API calls
```

If users need to call APIs provided by different cloud providers, they can combine the results of the API calls by calling our wrappers. Because the number of lines in JSON responses returned by API calls is reduced, users can switch between cloud providers. If users need to use services offered by all cloud providers, they can use the cloud provider that offers the most services. If users need to use services offered by only one cloud provider, they can use a different cloud provider that offers a unique service.

4.2.2 Language Used

We used Python to implement wrappers that call APIs provided by different cloud providers.

4.2.3 Tools Used

We used Microsoft's and Google's text processing APIs for our project. For the design of web Server Flask is used. The API responses are analyzed using Postman.

4.3 How to Generate Output Data

To generate output data, we would be creating JSON files with the text processing APIs from Google Cloud and Microsoft Azure using Python as our programming language. This would allow us to create and view various formats from these APIs. We would first input our desired text to be processed into the API call and would return us their own output data. With these generated output data, we would change the structure of the JSON file that generated and it would be used to create our own general format that would work with both Microsoft Azure and Google Cloud.

4.4 How to Test Against Hypothesis

To test our hypothesis, we would be mainly comparing the number of lines from the JSON file that we have generated with Google Cloud and Microsoft Azure's APIs and compare it with our new JSON format and the original JSON format from Microsoft and Google. We would only be comparing the API calls that both Google and Microsoft have, such as entity recognition and sentiment analysis. For the other API calls that are unique to the cloud service providers, we would include them in the Python file and use them normally to avoid vendor lock in.

5. Implementation

Various machine learning APIs like Sentiment analysis, Entity recognition, Content Classification, Entity Sentiment, Analysing Syntax, Entity linking, Entity PII, Key phase, Language detection offered by Google Cloud Platform and Microsoft Azure are used to implement the standardisation. A web server is designed using the Flask framework.

If an API is offered by a single cloud provider then the API result of that cloud provider is returned. If an API is offered by multiple cloud providers the APIs are standardized to a particular format and the user can choose a particular provider, then the result of that cloud providers API is converted to the standardized format and the result is returned. In the above mentioned APIs Entity Recognition and Sentiment Analysis APIs are offered by both Azure and GCP. So these APIs are standardized and returned whereas other APIs which are offered by a single cloud provider their results are returned as it is.

API endpoints for various APIs

API Endpoint	Cloud Provider/API
/api/entity-sentiment	GCP Entity-Sentiment API
/api/syntax	GCP Analyze syntax
/api/classification	GCP content Classification
/api/language_detection	Azure Language Detection
/api/pii_recognition	Azure Entity PII
/api/entity_linking	Azure Entity Linking
/api/key_phrase_extraction	Key Phrase
/api/entities	Entities based on Provider: GCP or Azure
/api/sentiment_analysis	Combined API results of GCP, Azure

Table 2: API endpoints for different APIs

5.1 Entity Recognition

We converted JSON responses to a general format for entity recognition. We would only select the features that both cloud service providers can offer and ignore the excessive features that one Cloud API call would have. In this case, we have chosen the features “name”, category“, “length”, and “offset” and this general format as a result would return an object with the common values returned by Google’s and Microsoft’s API calls for each entity.

```
from azure.ai.textanalytics import TextAnalyticsClient
from azure.core.credentials import AzureKeyCredential
import json

def convert_microsoft_entity():
    with open('entity_recog.json') as f:
        data = json.load(f)
    general_json = []

    for entity in data['entities']:
        new_entity = {}
        new_entity['name'] = entity['text']
        new_entity['category'] = entity['category']
        #new_entity['confidence_score'] = entity['confidence_score']
        new_entity['length'] = entity['length']
        new_entity['offset'] = entity['offset']
        general_json.append(new_entity)
    print(general_json)
    with open("general_microsoft_entity.json", "w") as outputfile:
        json.dump({'entities': general_json}, outputfile, indent=4)

def convert_google_entity():
    type_map = ['Unknown', 'Person', 'Location', 'Organization', 'Event', 'Work of Art',
                'Consumer Good', 'Other', 'Phone Number', 'Address', 'Date', 'Number', 'Price']
    with open('analyze_entities.json') as f:
        data = json.load(f)
    general_json = []
    #data = data[0]
```

```
    #print(data)
    for entity in data['entities']:
        new_entity = {}
        new_entity['name'] = entity['name']
        new_entity['category'] = type_map[int(entity['type'])]
        new_entity['length'] = len(entity['name'])
        new_entity['offset'] = entity['mentions'][0]['text']['beginOffset']
        general_json.append(new_entity)
    print(general_json)
    with open("general_google_entity.json", "w") as outputfile:
        json.dump({'entities': general_json}, outputfile, indent=4)
convert_microsoft_entity()
convert_google_entity()
```

5.2 Sentiment Analysis

We combined JSON responses into a general format for sentiment analysis. The general format returns a nested object with the values returned by Microsoft's API calls in the first object and the values returned by Google's API calls in the second object for each sentence.

Google's and Microsoft's API calls return scores for the same sentences. Google's API calls return the content. Microsoft's API calls return the offset and the length. The function sentiment adds the values returned by Microsoft's API calls to the nested object, adds the values returned by Google's API calls to the nested object, and finds the nested object for the sentence by calculating the offset.

```
import json

def sentiment():
        google_file = open("google.json", "r")
        microsoft_file = open("microsoft.json", "r")
        google_json = json.loads(google_file.read())
        microsoft_json = json.loads(microsoft_file.read())
        google_microsoft_json = {}
        google_microsoft_json["sentiment"] = {}
```

```
        sentences = []
#add the values returned by Microsoft's API calls to the nested object
        for sentence in microsoft_json["sentiment"]["documents"][0]["sentences"]:
                sentences.append({
                        "microsoft": sentence
                })
        i = 0
        offset = 0
#add the values returned by Google's API calls to the nested object
        for sentence in google_json[0]["sentences"]:
#find the nested object for the sentence by calculating the offset
                if (sentences[i]["microsoft"]["offset"] == offset):
                        sentences[i]["google"] = sentence
                offset += len(sentence["text"]["content"]) + 1
                i += 1
        google_microsoft_json["sentiment"]["sentences"] = sentences
        print(google_microsoft_json["sentiment"])

sentiment()
```

6. Data Analysis and Discussion

6.1 Output Generation

Note: This is not the entire JSON output. This output is just to show the format of the Microsoft and Google's APIs.

6.1.1 Original Format of Google's Entity Recognition API


```
{
   "entities": [
      {
         "name": "owner",
         "type": 1,
         "salience": 0.23477009,
         "mentions": [
            {
               "text": {
                  "content": "owner",
                  "beginOffset": 195
               },
               "type": 2
```

```
            }
          ],
          "metadata": {}
        }
        ...
    ],
    "language": "en"
}
```


6.1.2 Original Format of Google's Sentiment Analysis API


```
[
  {
    "sentences":[
      {
        "text":{
          "content":"We went to Contoso Steakhouse located at midtown NYC last week for a dinner party, and we adore the spot!",
          "beginOffset":-1
        },
        "sentiment":{
          "magnitude":0.8999999761581421,
          "score":0.8999999761581421
        }
      }
      ...
]
```


6.1.3 Original Format of Microsoft's Entity Recognition API


```
{
  "entities":{
    "documents":[
      {
        "id":"c3b55810-4282-47e2-b9f2-eae4e08802be",
        "entities":[
          {
            "text":"Contoso",
            "category":"Organization",
            "subcategory":null,
            "offset":11,
            "length":7,
            "confidencescore":0.58
          }
          ...
```

```
        ]
      }
    ],
    "errors":[

    ],
    "modelversion":"2021-01-15"
  }
}
```


6.1.4 Original Format of Microsoft's Sentiment Analysis API


```
{
  "sentiment":{
    "documents":[
      {
        "id":"c3b55810-4282-47e2-b9f2-eae4e08802be",
        "sentiment":"mixed",
        "confidenceScores":{
          "positive":0.86,
          "neutral":0.0,
          "negative":0.14
        },
        "sentences":[
          {
            "sentiment":"positive",
            "confidenceScores":{
              "positive":0.99,
              "neutral":0.01,
              "negative":0.0
            },
            "offset":0,
            "length":105
          }
          ...
        ]
      }
    ],
    "errors":[

    ],
    "modelversion":"2020-04-01"
  }
}
```

6.1.5 General Format of Standardized Entity Recognition API

```
{
  "entities":[
    {
      "name":"owner",
      "category":"Person",
      "length":5,
      "offset":195
    }
    ...
  ]
}
```

6.1.6 General Format of Standardized Sentiment Analysis API

```
{
  "sentences":[
    {
      "microsoft":{
        "sentiment":"positive",
        "confidenceScores":{
          "positive":0.99,
          "neutral":0.01,
          "negative":0.0
        },
        "offset":0,
        "length":105
      },
      "google":{
        "text":{
          "content":"We went to Contoso Steakhouse located at midtown NYC last week for a
dinner party, and we adore the spot!",
          "beginOffset":-1
        },
        "sentiment":{
          "magnitude":0.8999999761581421,
          "score":0.8999999761581421
        }
      }
    }
    ...
  ]
}
```

6.2 Output Analysis

	Number of Lines in Original Format	Number of Lines in General Format
Google's Entity Recognition API	396	154
Microsoft's Entity Recognition API	128	94
Google's Sentiment Analysis API	93	180
Microsoft's Sentiment Analysis API	99	

Table 3: Number of Lines after Generating the JSON Format

	Round Trip Time for Original Format	Round Trip Time for General Format
Google's Entity Recognition API	679 ms	561 ms
Microsoft's Entity Recognition API	1242 ms	1134 ms
Google's Sentiment Analysis API	655 ms	1587 ms
Microsoft's Sentiment Analysis API	1058 ms	

Table 4: Round Trip Time of API Calls

6.3 Compare Output Against Hypothesis

Our hypothesis is correct because the number of lines in JSON responses returned by API calls is less when users call our wrappers than when users call APIs provided by different cloud providers.

The number of lines in the general format of the standardized Entity Recognition API (154) is less than the number of lines in the original format of Google's Entity Recognition API (396). The number of lines in the general format of the standardized Entity Recognition API (94) is less than the number of lines in the original format of Microsoft's Entity Recognition API (128). The round trip time of the API call for the general format (561 ms) is less than the round trip time of Google's Entity Recognition API call for the original format (679 ms). The round trip time of the API call for the general format (1134 ms) is less than the round trip time of Microsoft's Entity Recognition API call for the original format (1242 ms). This is expected because the general format returns an object with the common values returned by Google's and Microsoft's API calls for each entity.

The number of lines in the general format of the standardized Sentiment Analysis API (180) is approximately the same as the sum of the number of lines in the original format of Google's Sentiment Analysis API (93) and the number of lines in the original format of Microsoft's Sentiment Analysis API (99). The round trip time of the API call for the general format (1587 ms) is less than the sum of the round trip time of Google's Sentiment Analysis API call for the original format (655 ms) and the round trip time of Microsoft's Sentiment Analysis API call for the original format (1058 ms). This is expected because the general format returns a nested object with the values returned by Microsoft's API calls in the first object and the values returned by Google's API calls in the second object for each sentence.

6.4 Abnormal Case Explanation

Although we have inputted the same text into the Google Cloud and Microsoft Azure's API, we can see that the Google's Entity Recognition API outputs a JSON file a lot larger than the Microsoft Azure's output and it may be due to their different implementations of how to recognize which word should be considered as an entity. They have used different machine learning algorithms to classify the words, and it is arbitrary to the cloud service provider or even the customers if they would like to include special categories, such as phone numbers and addresses to be considered as an entity. Therefore Google's entity recognition includes a lot more entities and additionally, Google also has included more description and variables about the entities as well to cause the JSON file to be larger than Microsoft's JSON file.

6.5 Discussion

Standardization Type	**Standardizing API responses and giving results of a particular provider**	**Giving the exact Response of the API of a particular provider**	**Combining responses of various cloud providers API and generating a response**
Cost	Charged for one Cloud Provider	Charged for one Cloud Provider	Charged for all the cloud providers who provide the API
API documentation	Have to go through a single documentation which gives the standardised results	Have to go through chosen cloud providers documentation to understand the API response	Have to go through multiple cloud providers API documentation to understand the API response

Table 5: Various Standardization Options and Analysis

We converted JSON responses to a general format for entity recognition because users can switch between a cloud provider if multiple cloud providers offer the service. Because the format of the API responses are similar, the general format standardizes the values returned by Google's and Microsoft's API calls. Both Google's and Microsoft's API calls return name, category, length, and offset values for each entity. The general format returns an object with the common values returned by Google's and Microsoft's API calls for each entity. The advantage of this format is that only one API needs to be called when converting JSON responses to a general format. The disadvantage of this format is that the values returned by the API calls need to be the same.

We combined JSON responses into a general format for sentiment analysis because users can switch between a cloud provider if multiple cloud providers offer the service. Because the format of the API responses are different, the general format combines the values returned by Google's and Microsoft's API calls. Google's API calls return numerical score and magnitude values for each sentence. Microsoft's API calls return categorical score values (positive, neutral, and negative) for each sentence. The values returned by Google's and Microsoft's API calls cannot be standardized because the scores will not be accurate if categories are converted to numbers. The general format returns a nested object with the values returned by Microsoft's API calls in the first object and the values returned by Google's API calls in the second object for each sentence. The advantage of this format is that the values returned by the API calls do not need to be the same. The disadvantage of this format is that multiple APIs need to be called when combining JSON responses into a general format.

We did not change JSON responses for unique API calls because users cannot switch between a cloud provider if only one cloud provider offers the service.

7. Conclusions and Recommendations

We wrote code that makes API calls compatible with different cloud providers, using machine learning APIs as an example. By standardizing the APIs the user can access multiple APIs from different cloud providers through one standardized API call which prevents vendor lock-in. In this project we designed and implemented a web server which provides results for various machine learning APIs by getting responses from multiple clouds which offer that API. We came up with three standardizing approaches. In the first approach, we standardize the JSON format of an API and translate the API response of the cloud provider the user prefers to the standardized JSON. In the second approach, we directly send the API response of the cloud provider the user prefers. In the third approach, we combine the results of all the JSON responses of various cloud providers which offer that API and return the modified JSON. We analyzed the number of lines required for the original API vs the number of lines required for the standardised API. We also analyzed round trip time for the original API response and the standardised API response. Our recommendations for future studies are standardizing other machine learning APIs, such as speech processing APIs, image analysis APIs, video analysis APIs, standardizing machine learning APIs provided by other cloud providers, such as Amazon and IBM, and standardizing other cloud APIs, such as APIs that provision resources for compute and storage services.

8. Bibliography

[1] Javier Miranda, Juan Manuel Murillo, Joaquín Guillén, and Carlos Canal. 2012. Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud SBAs. In Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups (WAS4FI-Mashups '12). Association for Computing Machinery, New York, NY, USA, 12–19. DOI:https://doi.org/10.1145/2377836.2377841

[2] R. Pellegrini, P. Rottmann and G. Strieder (2017). Preventing vendor lock-ins via an interoperable multi-cloud deployment approach. 2017 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, UK, 2017, 382-387, doi: 10.23919/ICITST.2017.8356428.

[3] Yasrab, R., & Gu, N. (2016). Multi-cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In. 2016 3rd International Conference on Information Science and Control Engineering (ICISCE), Information Science and Control Engineering (ICISCE), 2016 3rd International Conference on, Icisce, 473–477. https://doi-org.libproxy.scu.edu/10.1109/ICISCE.2016.108

[4] Comparing Machine Learning as a Service: Amazon, Microsoft Azure, Google Cloud AI, IBM Watson. Altexsoft. https://www.altexsoft.com/blog/datascience/comparing-machine-learning-as-a-service-amazon-microsoft-azure-google-cloud-ai-ibm-watson/.

[5] Quickstart | Cloud Natural Language API | Google Cloud. Google Cloud. https://cloud.google.com/natural-language/docs/quickstart.

[6] Text Analytics API Documentation - Tutorials, API Reference - Azure Cognitive Services - Azure Cognitive Services | Microsoft Docs. Microsoft Docs. https://docs.microsoft.com/en-us/azure/cognitive-services/text-analytics/.

9. Appendices

```
from flask import Flask
from flask import request
import json
from google.cloud import language_v1
from azure.ai.textanalytics import TextAnalyticsClient
from azure.core.credentials import AzureKeyCredential
key = "PASTE_YOUR_SUBSCRIPTION_KEY_HERE"   #Provide the Azure Key
endpoint = "PASTE_YOUR_COMPUTER_VISION_ENDPOINT_HERE" #Need to add endpoint

app = Flask(__name__)

def authenticate_client():
    ta_credential = AzureKeyCredential(key)
    text_analytics_client = TextAnalyticsClient(
            endpoint=endpoint,
            credential=ta_credential)
    return text_analytics_client



@app.route('/')
def hello_world():
    return {"message":"Hello, Call Machine Learning APIs and get the result from multiple clouds!!!"}
```

Figure 10: Source code of the web server

```python
#Google
@app.route('/api/entity-sentiment',methods=['POST'])
def entitySentiment():
    # print(request.get_json())
    content=request.get_json()
    data=content["text"]
    return analyze_entity_sentiment(data)

#Google
@app.route('/api/entities_gcp',methods=['POST'])
def entities_gcp():
    # print(request.get_json())
    content=request.get_json()
    data=content["text"]
    return analyze_entities(data)

#Azure
@app.route('/api/entities_azure',methods=['POST'])
def entities_azure():
    # print(request.get_json())
    content=request.get_json()
    data=content["text"]
    return entity_recognition_example(data)
```

Figure 11: URLs of APIs (entity-sentiment, entities)

```python
#Google
@app.route('/api/syntax',methods=['POST'])
def syntax():
    # print(request.get_json())
    content=request.get_json()
    data=content["text"]
    return analyze_syntax(data)


#Google
@app.route('/api/classification',methods=['POST'])
def classify():
    content=request.get_json()
    data=content["text"]
    return classify_text(data)

#Google
@app.route('/api/sentiment_gcp',methods=['POST'])
def sentiment_gcp():
    content=request.get_json()
    data=content["text"]
    return analyze(data)
```

Figure 12: URLs of APIs (Syntax, classification, sentiment)

```python
#Azure
@app.route('/api/sentiment_azure',methods=['POST'])
def sentiment_azure():
    content=request.get_json()
    data=content["text"]
    return sentiment_analysis_example(data)

#Azure
@app.route('/api/language_detection',methods=['POST'])
def language():
    content=request.get_json()
    data=content["text"]
    return language_detection_example(data)

#Azure
@app.route('/api/pii_recognition',methods=['POST'])
def pii_recognition():
    content=request.get_json()
    data=content["text"]
    return pii_recognition_example(data)
```

Figure 13: URLs of APIs (sentiment,language detection,pii recognition)

```python
#Azure
@app.route('/api/entity_linking',methods=['POST'])
def entity_linking():
    content=request.get_json()
    data=content["text"]
    return entity_linking_example(data)

#Azure
@app.route('/api/key_phrase_extraction',methods=['POST'])
def key_phrase_extraction():
    content=request.get_json()
    data=content["text"]
    return key_phrase_extraction_example(data)
```

Figure 14: URLs of APIs(entity_linking, key phrase)

```python
#Entities- Google and Azure
@app.route('/api/entities',methods=['POST'])
def entities():
    content=request.get_json()
    data=content["text"]
    provider=content["provider"]

    if(provider=="Azure"):
        entity_recognition_example(data)
        return convert_microsoft_entity()
    else:
        analyze_entities(data)
        return convert_google_entity()
```

Figure 15: URLs of APIs (Entities)

```python
@app.route('/api/sentiment_analysis',methods=['POST'])
def sentiment_analysis():
    content=request.get_json()
    data=content["text"]
    # provider=content["provider"]

    # if(provider=="Azure"):
    #     return sentiment_analysis_example(data)
    # else:
    #     return analyze(data)

    sentiment_analysis_example(data)
    analyze(data)
    return sentiment()
```

Figure 16: URLs of APIs (Sentiment analysis)

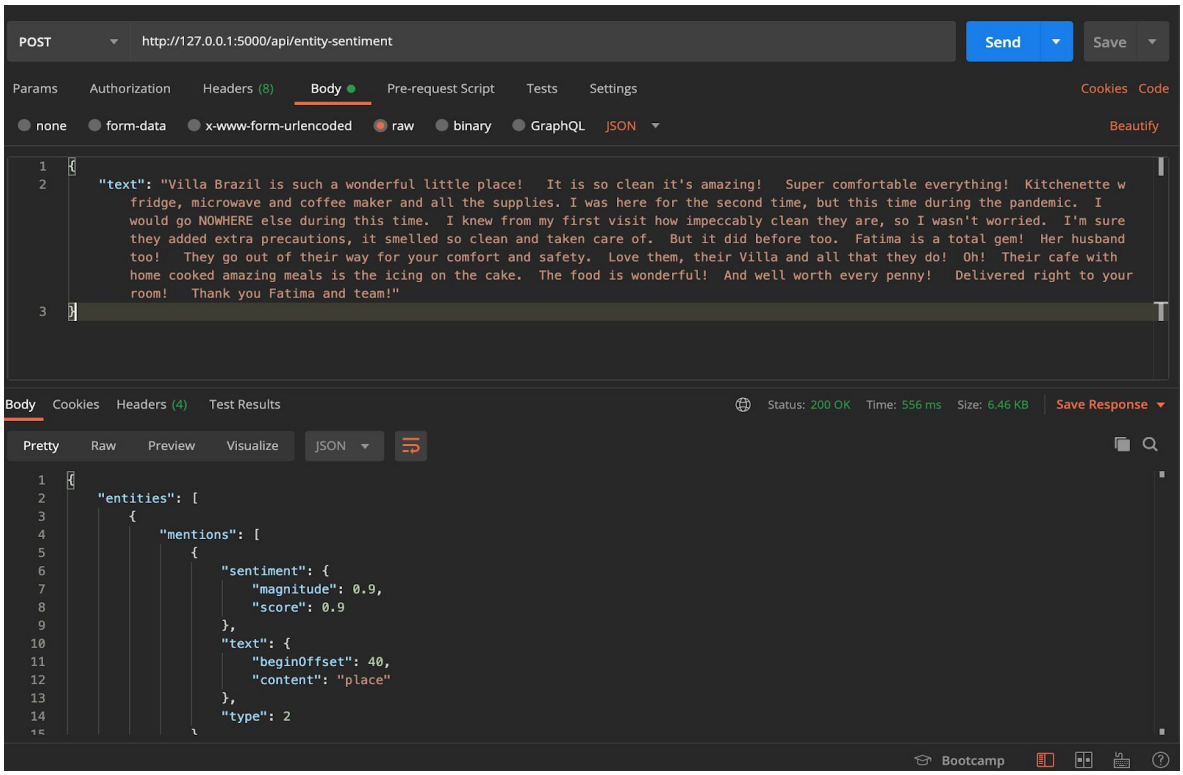

Figure 17: Entity-sentiment API call and json response

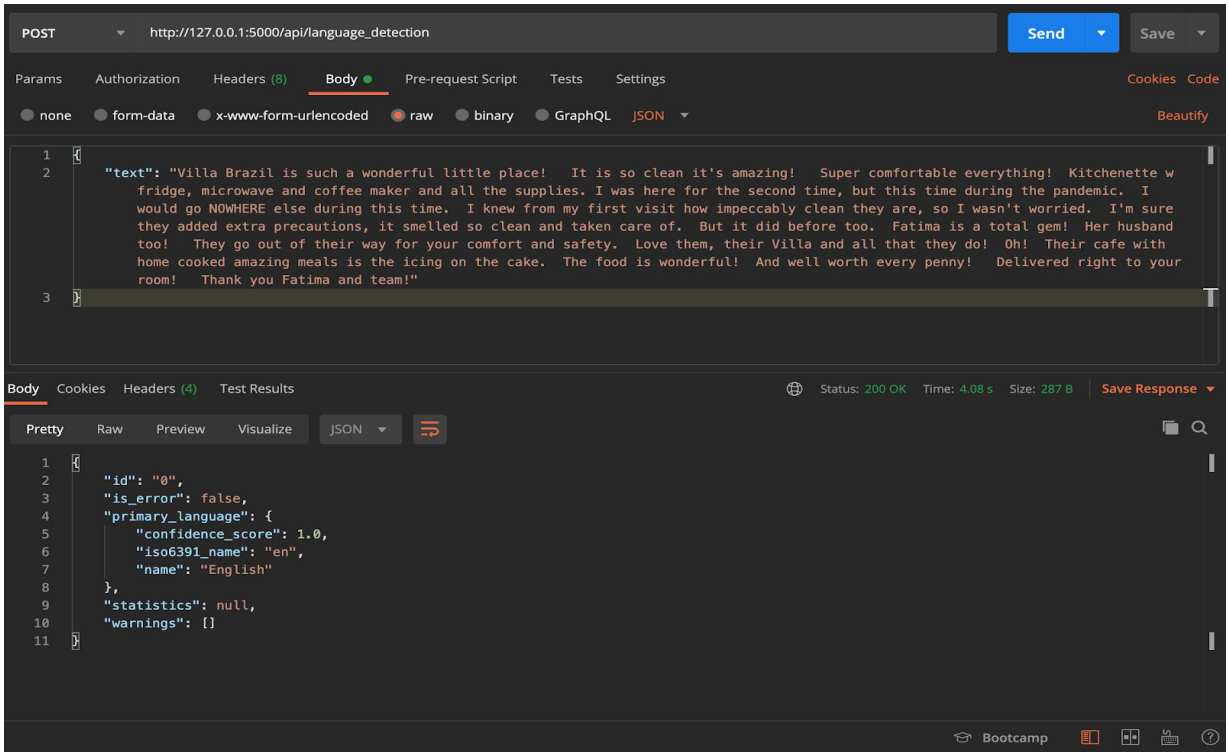

Figure 18: Language_detection API call and json response

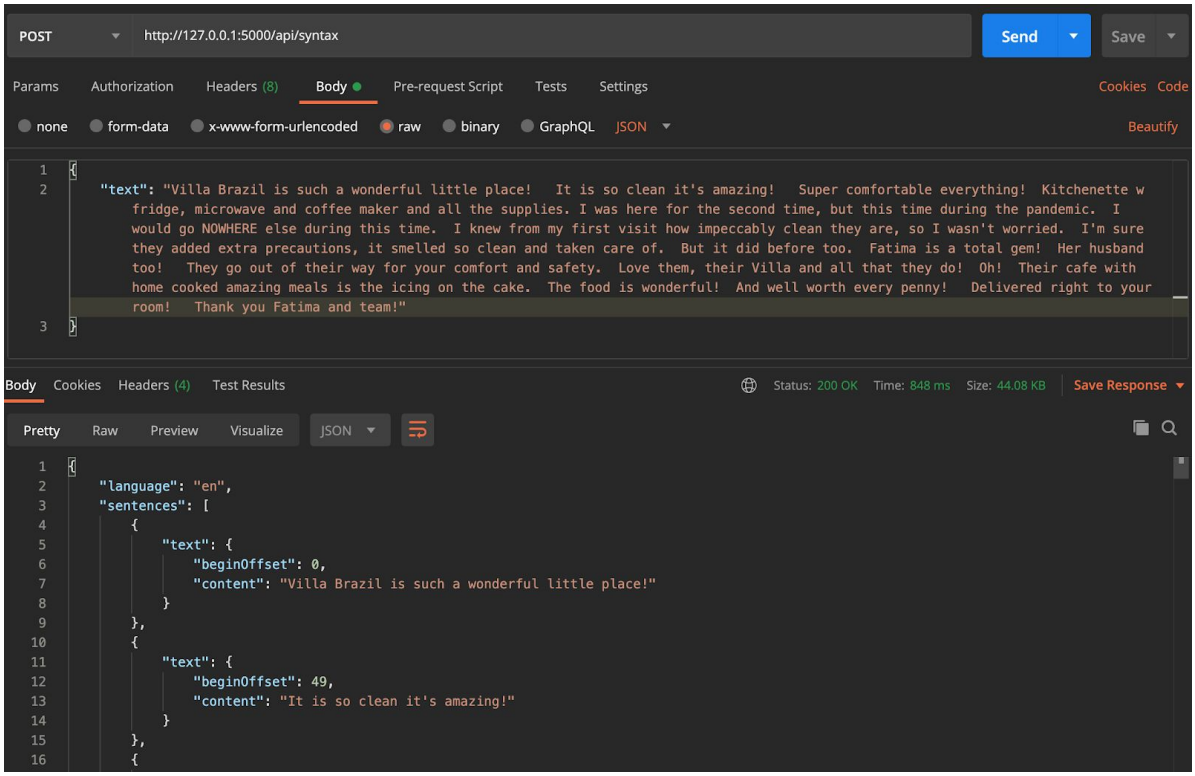

Figure 19 : Analyzing Syntax API call and JSON response

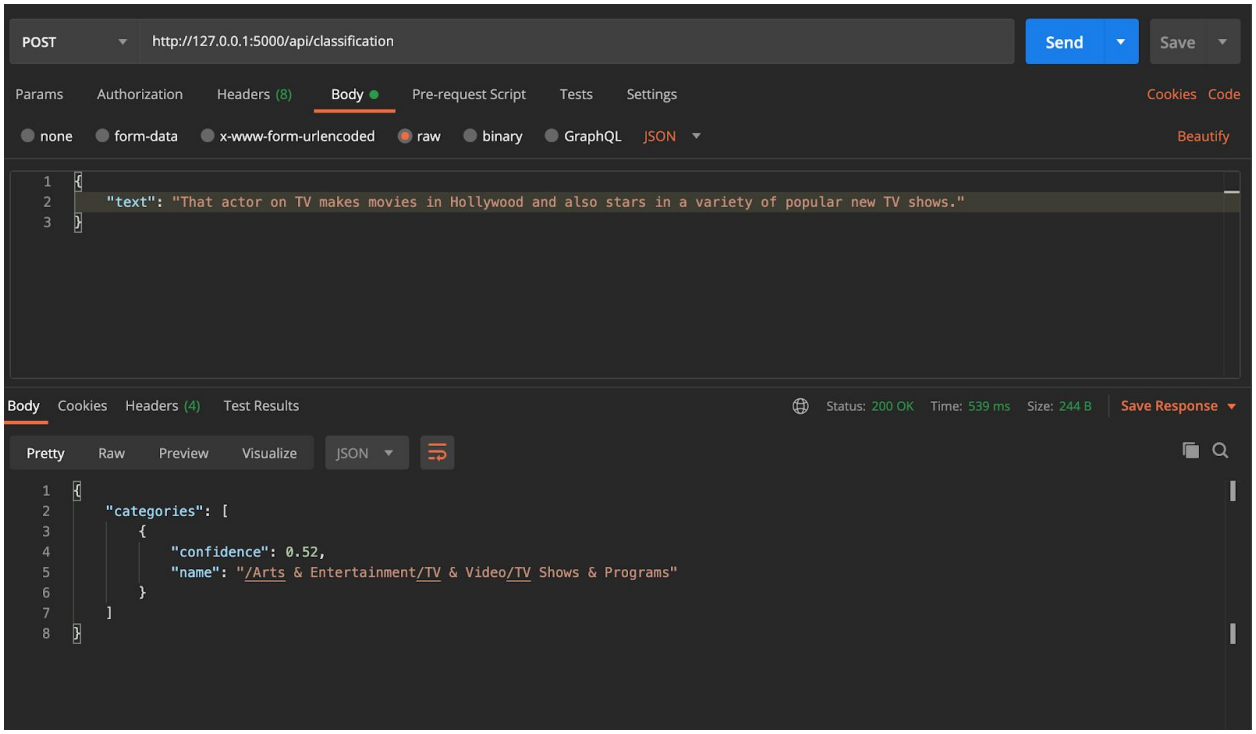

Figure 20: ClassifyText API call and JSON response

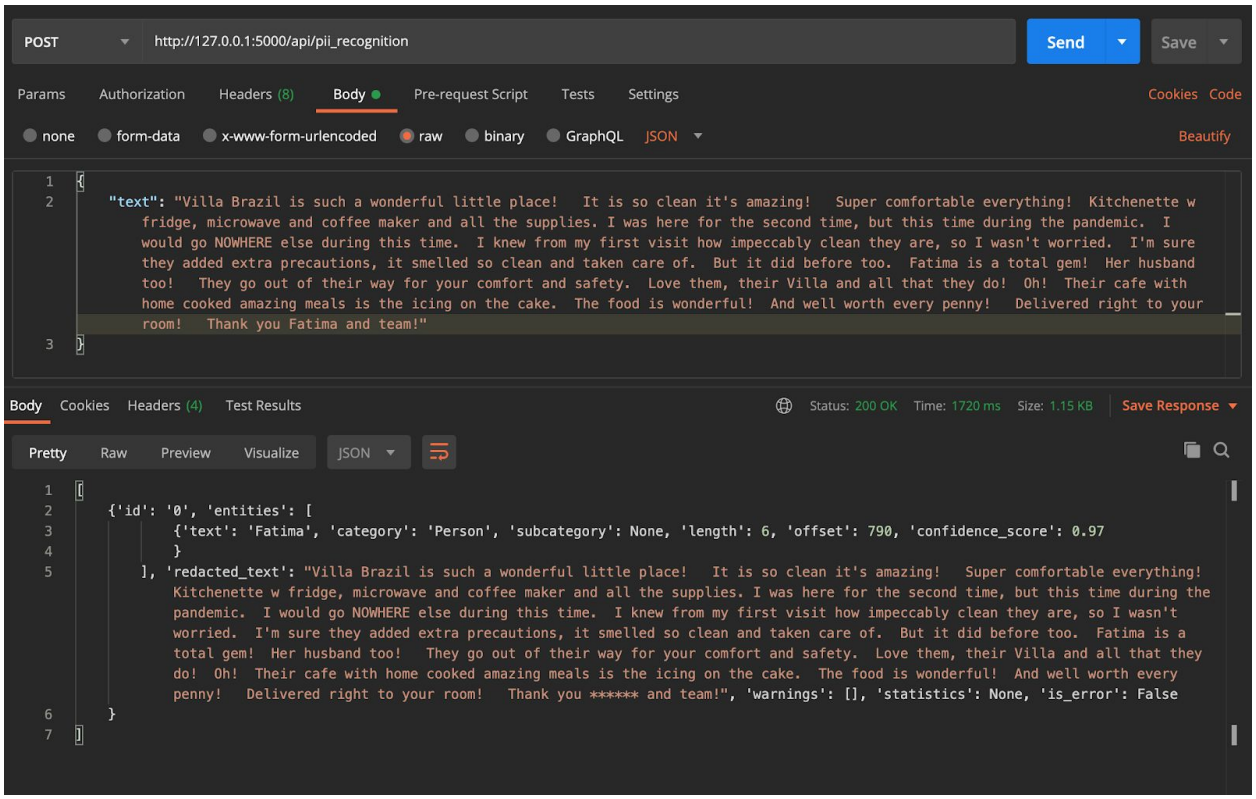

Figure 21: Entity PII API call and JSON response

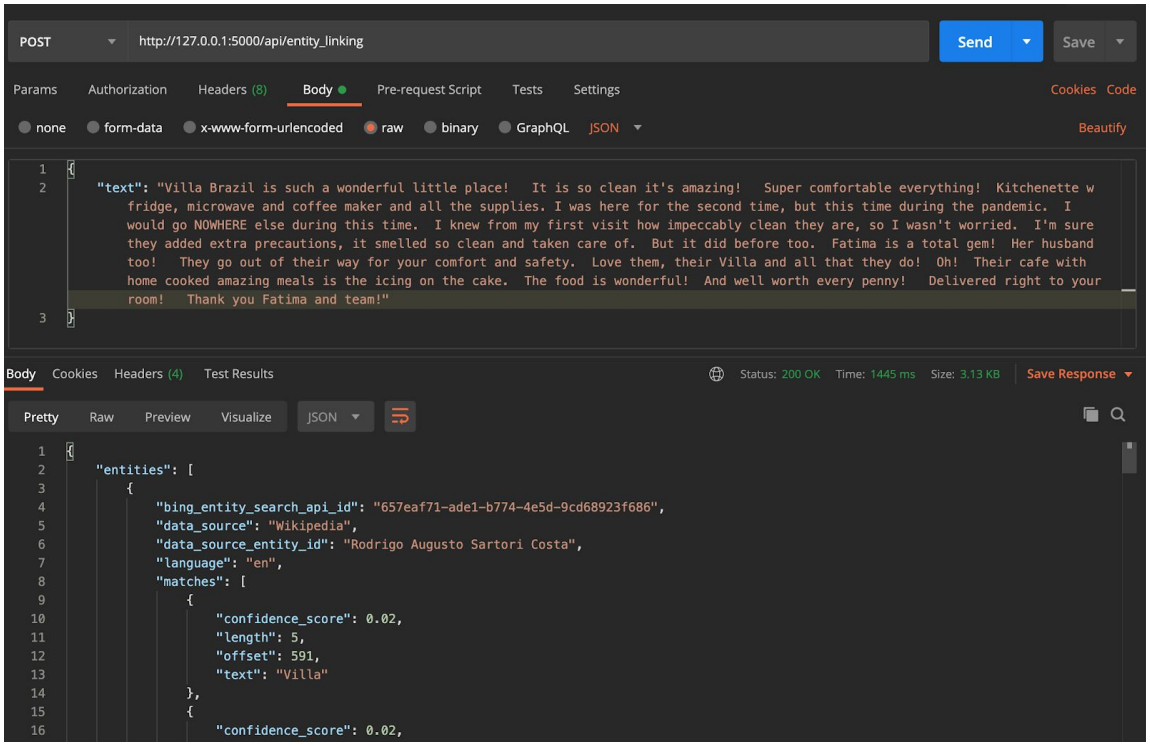

Figure 22: Entity Linking API call and JSON response

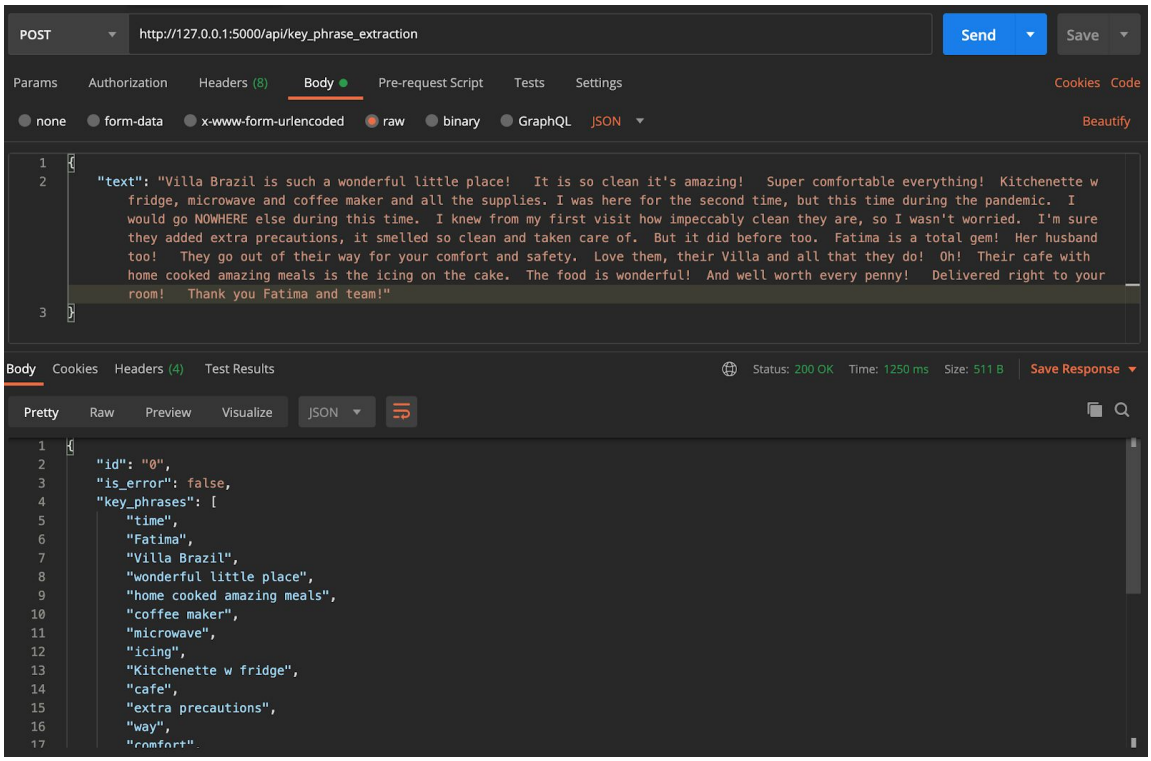

Figure 23: Key Phrase Extraction API call and JSON response

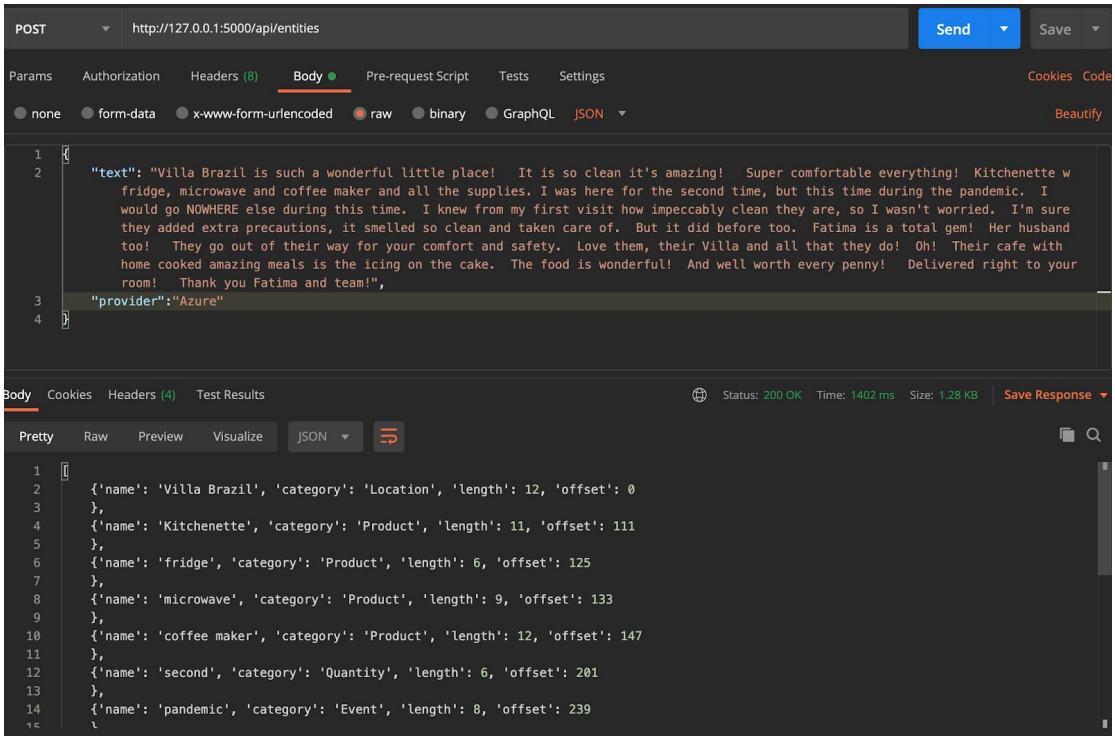

Figure 24: Entities API call with provider "Azure" to get results from Azure's Entities API

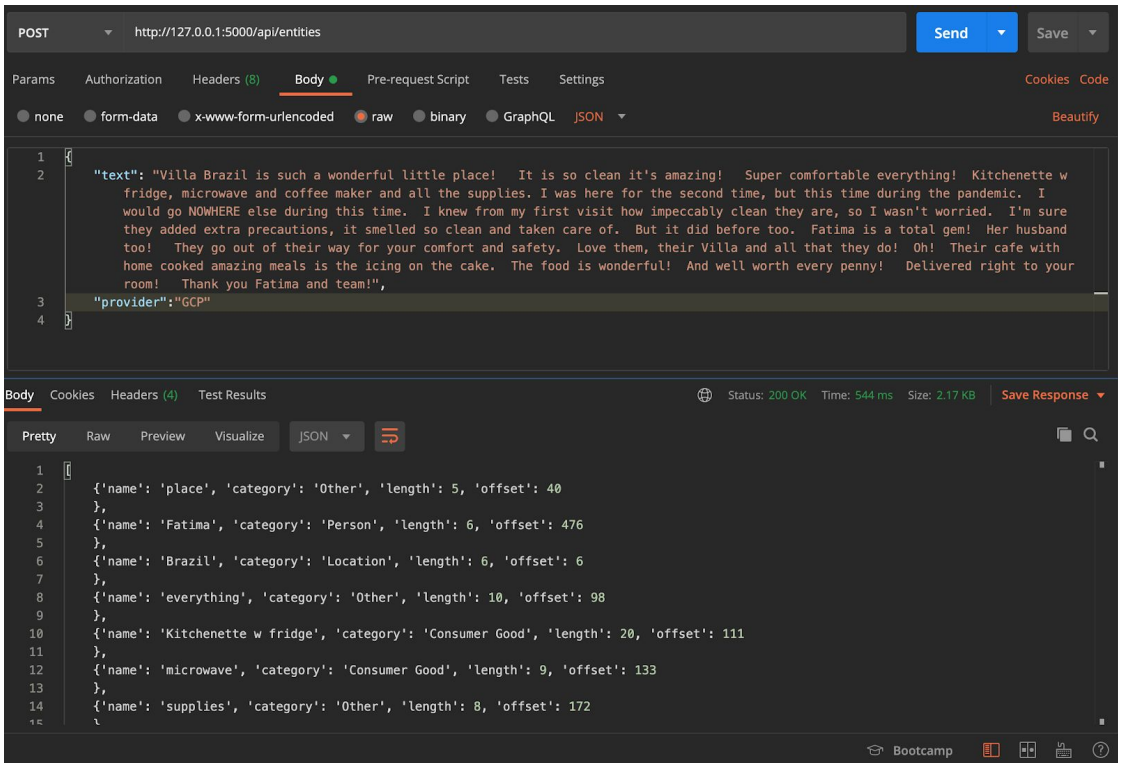

Figure 25: Entities API call with provider "GCP" to get results from GCP's Entities API

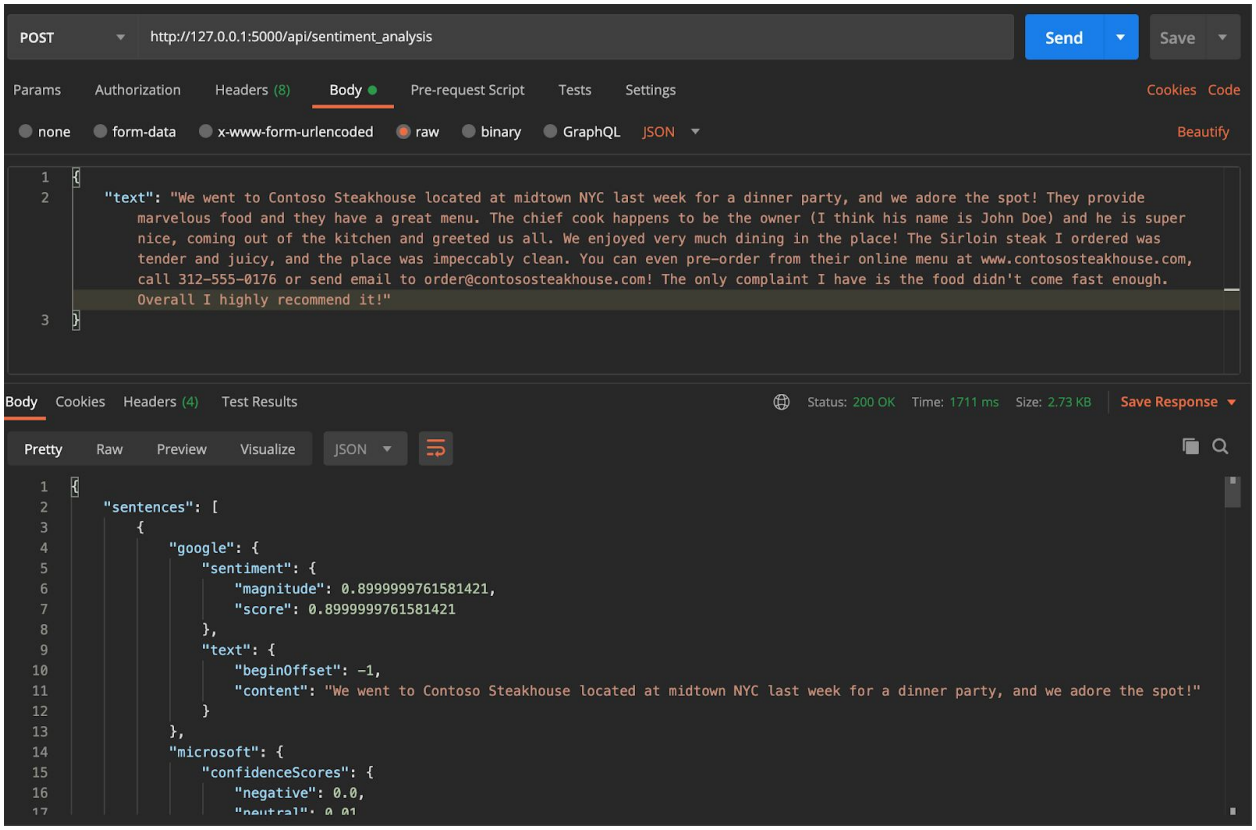

Figure 26: Sentiment Analysis API call with combined response of GCP and Azure API calls