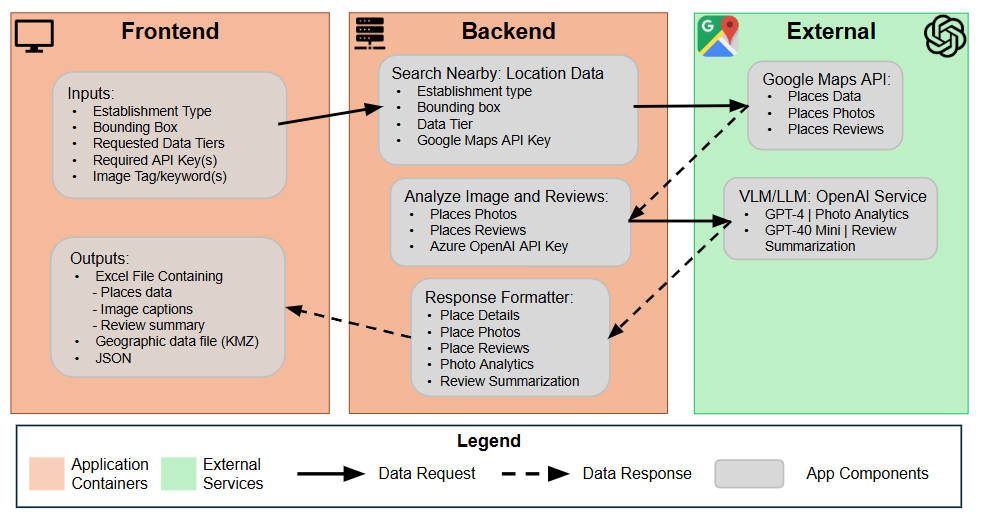
**SOF/C Tactical OPE Toolkit: User Guide**

**Introduction**

Special Operations Forces (SOF) analysts require critical information about locations of interest to conduct Operational Preparation of the Environment (OPE). Information necessary for their analysis includes: the name, address, hours of operation, relevant reviews, recent photos, and other attributes unique to each location. While this information is readily available, currently, there is no way for an analyst to query, process, and analyze this data quickly. The current workflow entails SOF analysts manually visiting the Google Maps website to extract data, spending 8-40 hours compiling the information into an Excel format document for each request. This process cannot take more than 24 hours. SOF analysts require an automated solution to efficiently gather and analyze critical details such as name, location, hours of operation, summary of reviews, and description of photos.

The Tactical OPE Toolkit is a web-based software application designed to automate the extraction and analysis of establishment data from Google Maps to support Operational Preparation of the Environment (OPE) for Special Operations Forces (SOF) analysts. It integrates the Google Maps Places application programming interface (API) and Azure OpenAI's services, which use the large language model (LLM) and vision language model (VLM) implementation, and outputs summarized intelligence in Excel and geo-referenced formats such as key markup language (KML) and key markup language zipped (KMZ). The manual approach took approximately 20 minutes per location, and the application can handle each location in about 75 seconds for a reduction in time of 94%.

**System Overview - High-Level Architecture**

****

Microservice Architecture: By adopting a microservice approach, the system ensures that each service (frontend, backend, and third-party integrations) is loosely coupled. This allows for easier updates, maintenance, and scaling as needed. The backend, which will handle external API calls and data processing, will be containerized using Docker and deployed in AKS. Using Kubernetes ensures that the application can scale horizontally as needed, with the ability to adjust resources dynamically based on traffic (Microsoft, n.d.).

Frontend: The frontend will be a Streamlit application, designed to be lightweight and user-friendly. It will handle minimal processing, primarily focusing on gathering and validating user input such as coordinates, bounding box, and ensuring the API keys are provided when a specific data tier is selected. After validation, the frontend will pass the data to the backend via a REST API, where all significant data processing will occur.

Backend: The backend will be built using FastAPI, which will handle most of the application's logic and external API interactions. The backend will be responsible for retrieving location data through the Google Maps API and processing this data. Additionally, the backend will interact with Azure OpenAI services, utilizing GPT-4o-mini to summarize reviews and GPT-4o to generate captions for relevant images. These tasks will be performed asynchronously to minimize response times, though minor latency is expected due to the nature of external service calls. Once the data is retrieved and processed, the backend will compile the results into a JavaScript Object Notation (JSON) object, which will be returned to the frontend. The frontend will then generate and provide users with downloadable output files, including an Excel spreadsheet containing the requested location information and a Keyhole Markup Language (KML) file for geospatial visualization.

**System Overview - Language Models**

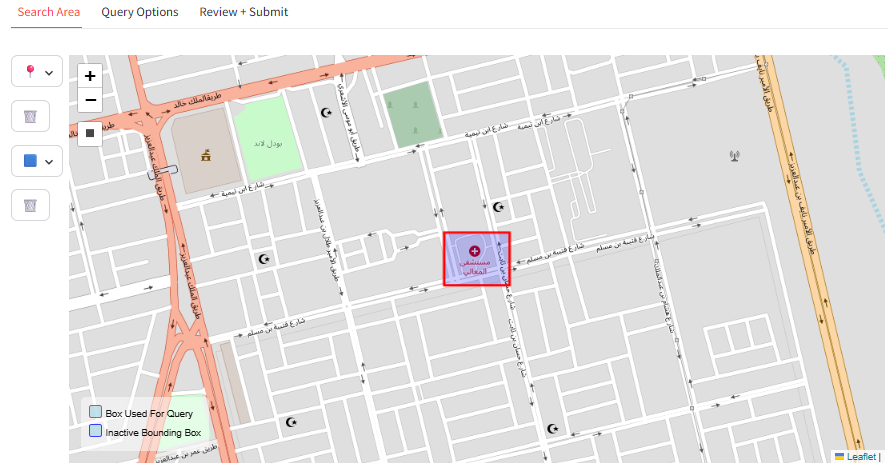
The application will leverage a GPT-40-Mini model for the LLM implementation. The GPT-40-mini model is a cutting-edge language model designed for efficient natural language processing tasks. It is optimized for low-latency applications while maintaining high-quality outputs. The model is beneficial for summarization, text generation, and conversational AI, making it an ideal choice for our application, which aims to summarize reviews from the Google Maps API. By leveraging GPT-40-mini, we can provide concise and meaningful summaries that enhance the user experience by distilling key insights from large volumes of text data.

This model was selected due to the customer's interest in exploring OpenAI solutions, a balance between computational efficiency and performance, and cost-effectiveness. Unlike larger models, GPT-40-mini requires less computational power, making it more cost-effective for deployment while still delivering accurate and coherent summaries. Additionally, its ability to handle various linguistic nuances ensures that summaries remain contextually relevant (OpenAI, n.d.).

The application will leverage a GPT-4 Turbo for the VLM implementation. This model is designed to process and analyze textual and visual data, making it an ideal choice for image understanding, object detection, and structured scene descriptions. GPT-4 Turbo was selected due to its high accuracy in vision-language tasks, its cost-effectiveness compared to larger models, and its capability to generate coherent, structured outputs for image analysis. The model is integrated into the system to process large-scale image datasets efficiently while maintaining low latency and scalability through Azure OpenAI Services.

**Using the System - Input Data Requirements**

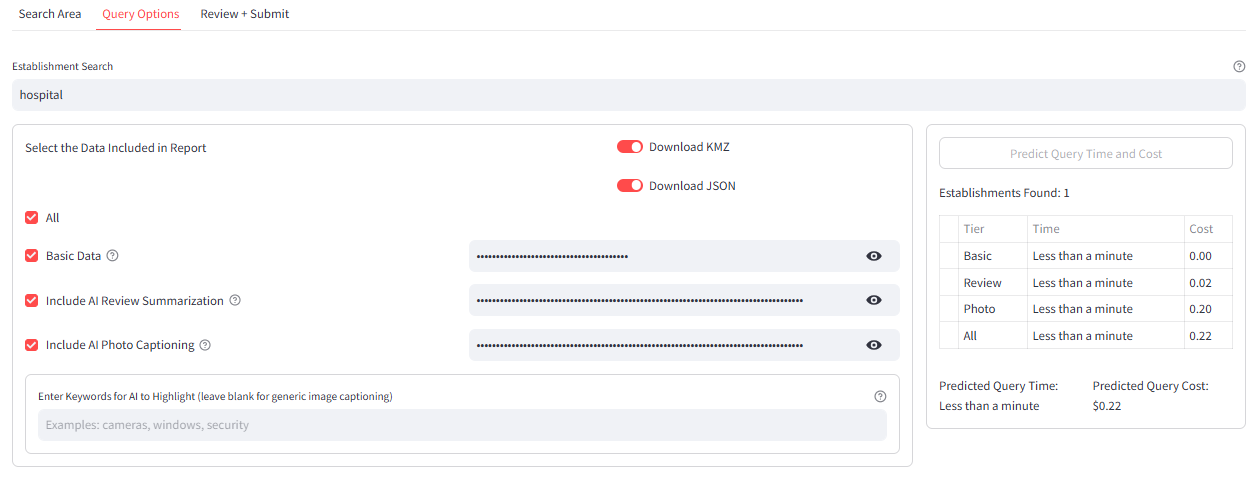
When users launch the application, they will land on the search area tab. Within the search area tab, the user is required to generate a bounding box. This can be achieved by entering the coordinates for the southwest and northeast corners of the bounding box or by drawing the bounding box on the embedded map. We have also enabled a drop pin function, allowing users to enter the latitude and longitude coordinates and drop a pin on the embedded map to streamline their ability to draw a bounding box.



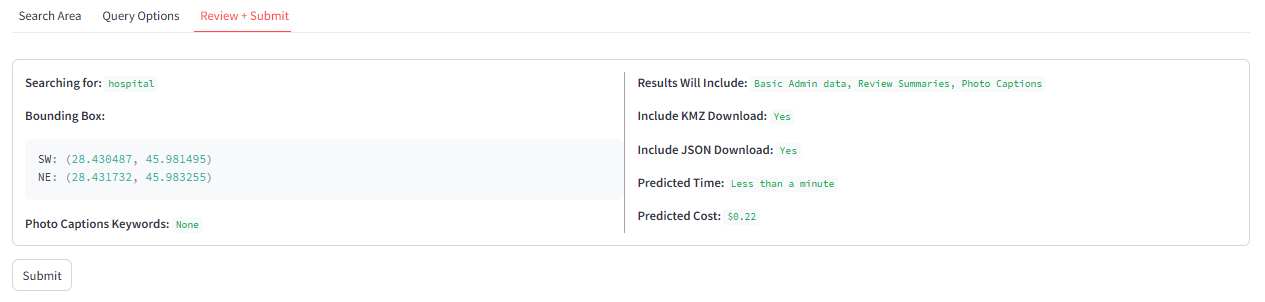
On the query options tab of the application, the user is required to provide an establishment search term (examples: food, hotels, gas stations) and select the data included in the output as well as the output types. The user will check the appropriate boxes to signify the data tier they desire; the data tiers are as follows:

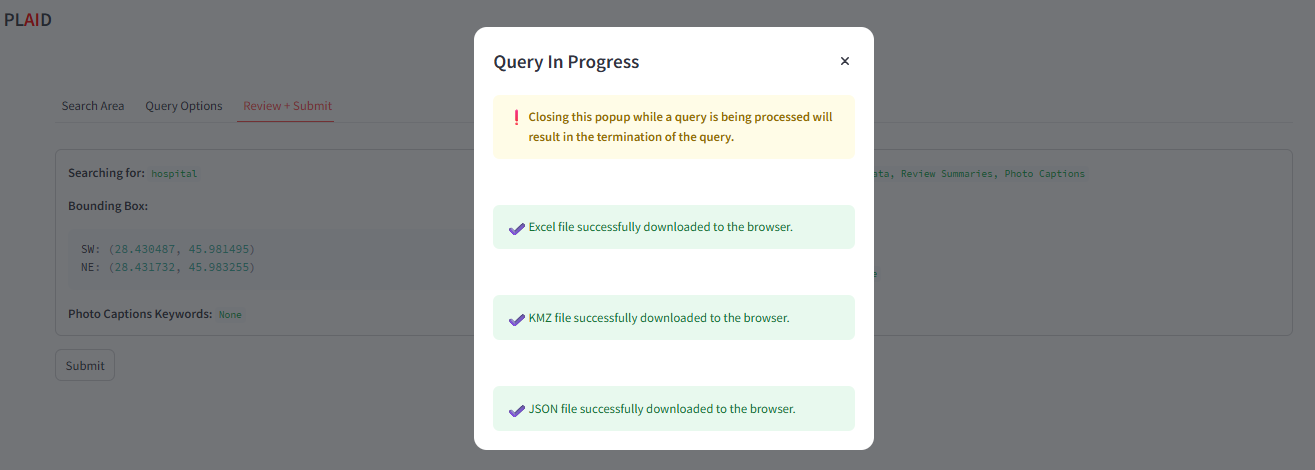
* Basic data
* Include AI review summarization
* Include AI photo captioning

The basic data tier is always selected by default. When the user selects the reviews or photos data tiers, they will be required to provide their API key for the specific service. The basic data tier requires the user's Google Maps API key, and the review summarization and photo captioning require the user's Azure OpenAI Service key. If the user selected the “include AI photo captioning” data tier, an optional field will be displayed prompting the user to enter keywords for the VLM to highlight. If the user does not provide input to the keywords for the “AI to highlight” field, the VLM will process the images with a neutral prompt and not focus on anything specific. If the user provides keywords, they will be integrated into the system message provided to the Azure OpenAI service. This will ensure that any keywords listed will be annotated and described if present in the picture or identified as not present in the pictures. The user will also identify if they want the results displayed in a KMZ or JSON file, along with the default Excel file. The user can also select the “predict query time and cost” button, which will provide the number of locations that match the search term and are within the bounding box. Moreover, it will estimate the cost and time associated with each data tier and with all data tiers selected.



The review and submit tab will display the parameters to the user to validate the input fields before submitting the query to the selected APIs. If the user is satisfied with the query, they will select submit, and the submit button will display a pop-up window stating “query in progress”. The user cannot click out of the pop-up window or it will cause the query to terminate. The user will know that the query is complete when the pop-up window states that each of the user-selected outputs have been successfully downloaded to the browser. The user can validate this by checking the downloads section of their internet browser.





The input requirements from the UI will generate a JSON based on the user inputs for the backend to process and provide to the required API services. Below is an example of the JSON that would have been created based on the input fields shown in the application screen captures above.

{

"text\_query": "hospital",

"lat\_sw": 28.430487,

"lng\_sw": 45.981495,

"lat\_ne": 28.431732,

"lng\_ne": 45.983255,

"prompt\_info": "",

"tiers": ["reviews", "photos"],

"google\_api\_key": REDACTED,

"llm\_key": REDACTED,

"vlm\_key": REDACTED

}

**Using the System - Model Invocation**

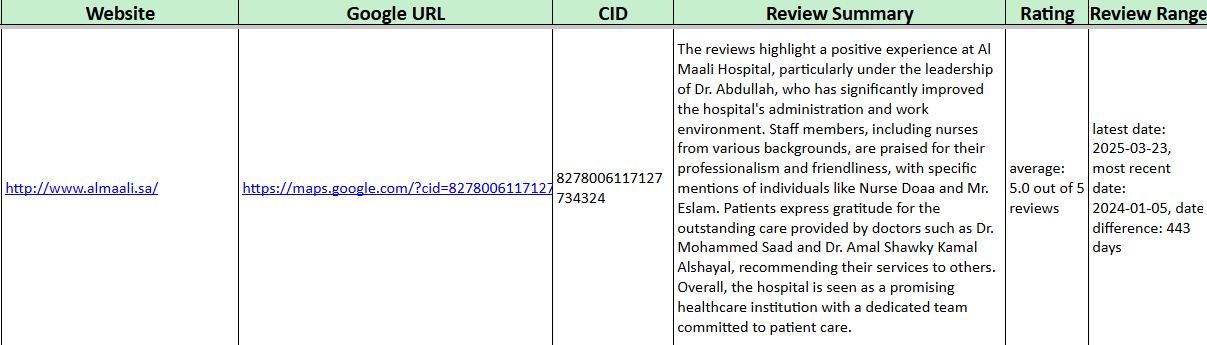
The LLM used for review summarization and the VLM used for photo captioning are invoked when the user submits a query with the reviews or photos data tiers selected from the UI. Related to the model invocation, the backend is equipped with an API service, an API service helper function, LLM, and VLM Python files. The backend handles the JSON by using conditional logic to identify if the data tier was selected and if the keys are present in the JSON. The llmservice.py can be found in Appendix 1, and the vlmservice.py can be found in Appendix 2.

**Using the System - Output Interpretation**

The output from the VLM, LLM, and Google Maps API is all JSON files. The backend has a formatting function that cleans the data, enabling efficient loading of the JSON into the Excel and KMZ file formats. The user will not be required to handle the outputs in JSON format; they will interact with the data in Excel and KMZ formats per the project specifications. The user will have an option to download the output results in a JSON format to enable integration with retrieval augmented generation (RAG) LLM solutions. The first sheet of the Excel file is named locations and will contain all the locations identified within the query. This sheet is designed to serve as a one-stop shop for the user and provide them with the Google Maps API data as well as a summarization of the review and photos if those data tiers are selected. The locations sheet of the Excel file has the following columns:

* Name (original)
* Name (translated)
* Address
* Phone
* Latitude
* Longitude
* Hours
* Website
* Google URL (link to location on Google Maps)
* CID (unique identifier provided by Google Maps)
* Review Summary (summary of the reviews for the location)
* Rating
* Review Range (provides the latest date and the most recent dates of the reviews and the difference between)
* Summary of Image (summary of the image captions for the location)
* Photos URL (link to Google Photos of the location)
* Review Link (hyperlink to the Excel Sheet for the locations reviews)
* Images Link (hyperlink to the Excel Sheet for the locations' images)

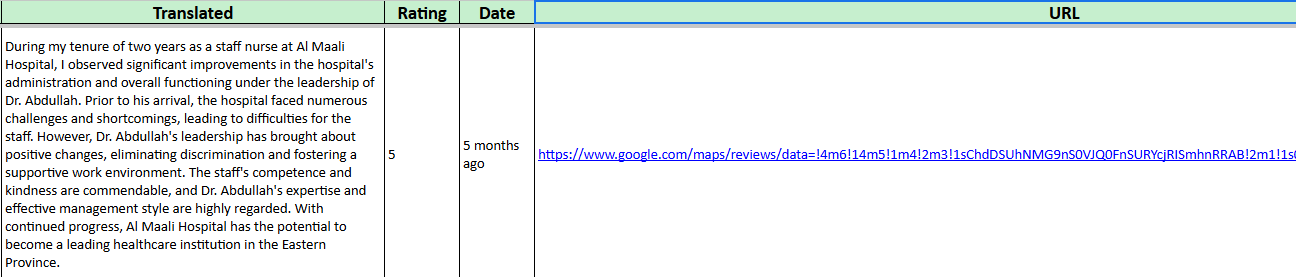
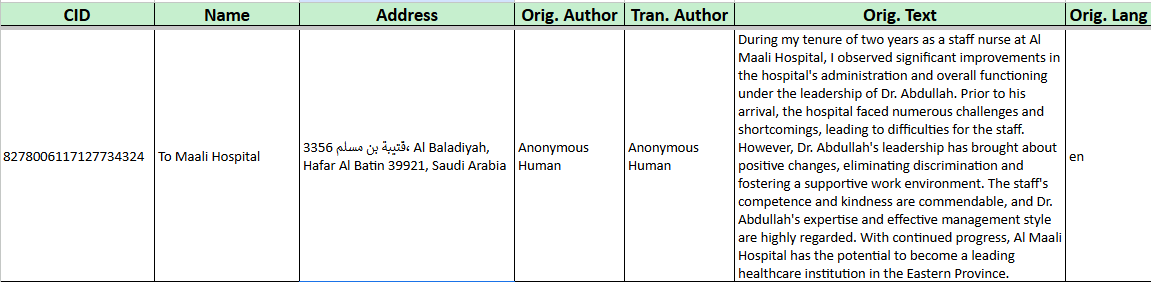
The application uses the Google Maps API Text Search endpoint, which has a limit of 60 places being retrieved per API call. The places that are returned are those that meet the user's input requirements and those deemed most relevant by Google. For example, if you search for a coffee shop in all of New York City, you would only get 60 results, even though there are more than 60 coffee shops in the bounding box. Moreover, it is unlikely that you will get the same results every time or get completely new results if you run the same query multiple times. Below is an example of the Excel output from the query outlined in the input data requirements section.



Following the location sheet in the Excel file, there will be one review and one image sheet for each location. While this can amass a large number of sheets, this underpins why it is important to use the hyperlinks in the locations sheet. In the review sheet, the name is a composite of the location name and the CID. This technique was developed to ensure a unique name for each sheet. So if there are several locations in a query with the same name, now users can differentiate between them. The columns in the review sheet are as follows:

* CID
* Name
* Address
* Original Author
* Translated Author
* Original Text
* Original Language
* Translated
* Rating
* Date
* URL

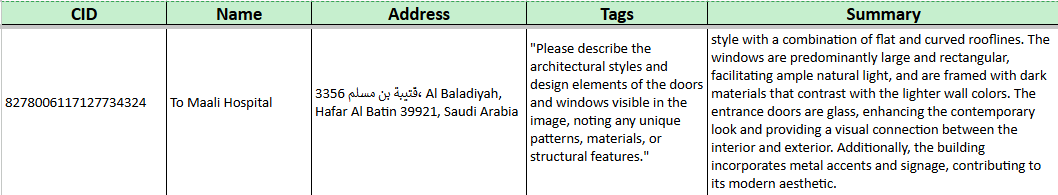
The Google Maps API, using the text search endpoints, limits the number of reviews to five per API call. Based on the Google Maps API documentation, the reviews we received are deemed the most relevant by Google. The application uses the Google Translator Python library to detect and provide the translation. Below is an example of the review sheet related to the query executed in the input data requirements section. In the example, the translation column was not necessary because the original text of the review was in English.



The images sheet within the Excel output displays each of the ten photos and one street view photo collected by the Google Maps API. The endpoint has a limit of ten photos per location and one street view photo per API query; the returned photos are deemed most relevant by Google. The images are not provided to the user, but the Excel output has a link to the image. The application only handles the images in binary and converts them to base64 for processing by the VLM. This method significantly reduced the application’s processing time. The images sheet has the following columns:

* CID
* Name
* Address
* Tags (prompt provided to the VLM, which contains the user's keywords from the UI)
* Summary (VLM image caption)
* Link (URL to the specific image)

Below is an example of the images sheet related to the query executed in the input data requirements section.



**Deployment and Integration - Deployment Options**

The application is available in two versions to meet the customer’s operational requirements: a cloud-based deployment within Microsoft Azure and a standalone version. The solution is containerized, comprising separate frontend and backend components to support modularity and scalability across deployment environments. The details to deploy the standalone version can be found in Appendix 3. The dependencies for the deployment can be found in Appendix 4. The steps for the cloud deployment are outlined below:

* Environment Setup: The infrastructure environment will be established using a Terraform configuration file, enabling future customers to replicate the deployment environment with minimal effort. This configuration will provision the AKS cluster, Resource Group, and ACR within the East USgeographical region.
* Configuration AKS: Upon provisioning the AKS cluster, deployment and service YAML files will be generated for both the frontend and backend containers. Additionally, HTTPRoute and Gateway YAML files will be created to configure the API gateway. This setup will allocate a public IP address and initiate internal routing to the frontend container.
* Configuration ACR: The Azure Container Registry (ACR) will be provisioned with a managed identity of ACRPULL to enable the AKS cluster to pull images from the registry. It will be configured using the Basic tier to optimize cost efficiency. We will use Github repo and define a workflow to be triggered upon a merge of the main branch. This yaml will allow the new image to be created, pushed to the ACR, and then applied to the Kubernetes cluster.
* Deployment Scripts: The infrastructure will be provisioned using a Terraform configuration file to ensure reproducibility. Additionally, the YAML files for the AKS cluster, along with the Terraform scripts, will be hosted on a Github repository to allow the customer to reference and use for their deployment.
* Testing and Validation: To accurately test the deployment we will use the domain name to access the application and run sample queries. These queries should return the reviews, images, and data that was requested. We will run a minimum of five queries to ensure accuracy, and obtain a baseline CPU and memory utilization.
* Rollback and Version Control: Github will be used as a version control system and CI/CD system. There are branches for frontend and backend which will be merged into the main branch which will be used for deployment. A pipeline will be defined so that once a merge occurs to the main branch it will be uploaded to the ACR.

**Deployment and Integration - Integration with Existing Systems**

The application does not include a machine learning model but leverages a LLM for review summarization and a VLM for image captioning. The application interacts with the Azure OpenAI LLM and VLM through API calls from the backend. Examples of the LLM and VLM python files can be found in Appendix 1 and 2 respectively. Moreover, the libraries used Azure OpenAI API can be found in Appendix 4 under the backend dependencies section.

**Monitoring and Maintenance**

The infrastructure is deployed on Microsoft Azure Kubernetes Service (AKS), utilizing two initial pods within the cluster. To evaluate performance under load, the application was vertically scaled using a locust.py script and the Apache JMeter application, simulating 10 concurrent users executing queries. Based on the results of this testing, an assessment was conducted to determine the number of pods required to support this user load efficiently, leading to the implementation of horizontal pod autoscaling.

Following the scaling implementation, it was determined that monitoring capabilities were necessary to ensure the application performs as expected within the cluster. For the cloud deployment, Azure’s managed Prometheus solution was selected to collect time-series metrics and generate performance alerts. Additionally, the managed Grafana service will be utilized to provide dashboards and visualizations based on the data collected by Prometheus.

To monitor the standalone application, users can utilize Docker Desktop to inspect the status of running containers. By accessing the container view and selecting the three-dot (ellipsis) menu, the **"View Details"** option provides access to logs, resource usage statistics, and image inspection. This interface enables users to assess container health, identify potential resource overcommitment, and troubleshoot issues within the standard deployment environment.

Since the AI model will utilize a managed service, retraining or manual updates will not be necessary. The only requirement is to ensure that the model remains within its support lifecycle and is not approaching retirement. Responsibility for monitoring this will fall to either AI2C or the organization deploying the application.

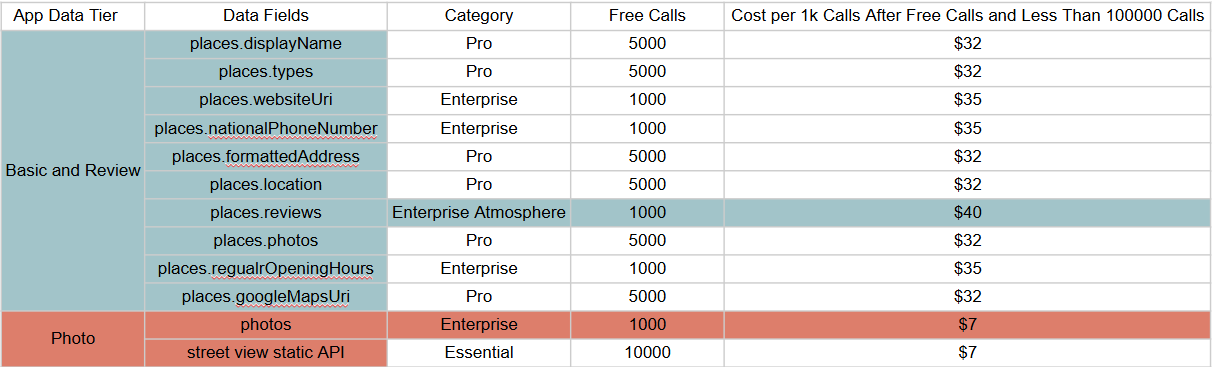
The Google Maps API and Azure OpenAI API websites both provide dashboards to monitor spending, token usage, and many additional metrics. The user is responsible for obtaining their own APIs subscriptions and funding the token used by the API each month. The Google Maps API dashboard can be found at the following URL: (https://console.cloud.google.com/apis/dashboard). The Azure OpenAI Service dashboard can be found under the cost management section of the Azure portal.

**Common Issues and Solutions**

The most common issues associated with the application are not within the application itself but with leveraging Google Maps and Azure OpenAI API services. The Google Maps Text Search New API Endpoint has a limit of only 60 locations that can be returned in one API call. Moreover, when we called specific field masks within the endpoint, you are only able to retrieve five reviews and ten photos per API call. We are unable to develop solutions to circumvent these limitations. However, to enable the user to retrieve more than 60 locations in a specific area, we added the bounding box to the KMZ output and the cost and price prediction capability which identifies the number of locations in the bounding box that match the search term. These two functions coupled together will enable the user to create smaller bounding boxes and run several queries to cover the original larger bounding box.

For example, the user wants to search a 16 square block area for coffee shops. There time and cost predictor informed the user there are 60 coffee shops in their bounding box. This indicates that the user might have exactly 60 or more likely that there are more than 60 coffee shops in the bounding box. The user can now break the 16 square block area into multiple queries, testing the total number of coffee shops in each bounding box until it is less than 60. The simplest application would be to divide the 16 square block area into two and if that bounding box returns 60 coffee shops then separate the 16 square block area into four queries. The user will be able to visualize the bounding box of each query in Google Earth using the KMZ and determine the bounding box coordinates for the subsequent query.

**Cost Breakdown - Google Maps API**

****

The above chart outlines the application data tiers, the data fields called within the application, the category determined by Google along with the free calls and cost per 1,000 calls after the free calls capacity has been reached but less than 100,000 calls. The blue section outlines the columns associated with the basic and review data tiers and red is associated with the photos data tier.

When a user executes a Basic and/or Review query, there is a free capacity of 1000 calls. Each query by the user will trigger 1-3 calls based on how many locations are in the query. One call returns up to 20 locations and each subsequent call returns an additional 20 locations not to exceed 60 total locations. Each month, the user will get about 330 - 1000 queries for free. The customer estimated they would only query approximately 400 locations per month, thus they would not exceed their free capacity for basic and review queries.

The users queries against the photos data tier include an additional API call to get the binary of each photo. Per month, the user will get 1000 photos for free. Based on the customers estimate of approximately 400 locations per month we expect them to exceed the free tier capacity of 1000 calls if each location has 10 photos (limit of text search new endpoint). Thus, each additional photo processed by the application over the 1000 calls will cost $0.007.

**Cost Breakdown - Azure OpenAI Services**

The cost for the Azure OpenAI Service is entirely dependent on the token usage. The application uses a GPT-40-mini and the LLM and the GPT-4 Turbo for the VLM to reduce cost while maximizing performance and speed. The Google Maps API Text Search New Endpoint only allows the user to return five reviews, ten photos, and one street view photo per query. Through our testing we estimated and validated that the five reviews cost about $0.01 and the ten photos along with the street view photo cost about $0.03. Based on the customers estimate of approximately 400 locations per month, we expect them to spend $16 on Azure OpenAI services.

### **Glossary of Terms**

1. **AKS (Azure Kubernetes Service)**A managed Kubernetes service provided by Microsoft Azure that simplifies deploying, managing, and scaling containerized applications using Kubernetes.
2. **API (Application Programming Interface)**A set of rules that allow different software applications to communicate with each other. In this application, APIs are used to retrieve location data from Google Maps and interact with OpenAI's language models.
3. **Azure OpenAI Services**Microsoft Azure's platform for accessing OpenAI's models, such as GPT-4o and GPT-4o-mini, for language and vision tasks.
4. **Backend**The server-side component of the application responsible for processing logic, handling API calls, and interacting with external services. It is built with FastAPI.
5. **Bounding Box**A geographic area defined by two sets of latitude and longitude coordinates (southwest and northeast corners) used to specify the search area for location data.
6. **CID (Customer ID)**A unique identifier assigned by Google Maps to each location or place.
7. **Docker**An open-source platform used to containerize applications, ensuring they run reliably across different computing environments.
8. **Excel**A Microsoft Office application used for creating spreadsheets; in this toolkit, it is the default format for presenting extracted location data.
9. **FastAPI**A modern Python web framework used for building APIs quickly and efficiently, known for high performance and ease of use.
10. **Frontend**The user interface of the application, developed using Streamlit, where users input search criteria and download results.
11. **GPT-4 Turbo**An advanced vision-language model (VLM) from OpenAI that can analyze and caption images based on user-provided prompts or general content understanding.
12. **GPT-40-mini**A lightweight, efficient language model from OpenAI optimized for summarizing large volumes of text, used for condensing Google Maps reviews in this application.
13. **JSON (JavaScript Object Notation)**A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is used internally to structure input and output data.
14. **KML (Keyhole Markup Language)**An XML-based format used to represent geographic data in applications like Google Earth.
15. **KMZ (Keyhole Markup Language Zipped)**A compressed version of a KML file that bundles the KML along with associated resources for easier sharing and loading.
16. **LLM (Large Language Model)**A type of AI model trained to understand and generate human-like text. In this toolkit, it is used for summarizing reviews.
17. **Locust.py**An open-source load testing tool used to simulate multiple users accessing the application to test its performance under load.
18. **Microservice Architecture**A design pattern where an application is built as a collection of loosely coupled, independently deployable services.
19. **Operational Preparation of the Environment (OPE)**Activities conducted to analyze and shape the operational environment before conducting missions, commonly used by Special Operations Forces.
20. **Prometheus**An open-source monitoring and alerting toolkit used for collecting and querying metrics from containers and services.
21. **REST API (Representational State Transfer API)**An architectural style for building web services that interact via HTTP requests for operations like retrieving and updating data.
22. **Streamlit**An open-source Python library used to create web apps for data science and machine learning projects quickly and easily.
23. **Terraform**An open-source infrastructure as code software tool that provides a consistent CLI workflow to manage cloud services.
24. **VLM (Vision Language Model)**An AI model capable of understanding and describing visual inputs such as images. In this application, it captions photos from Google Maps.

## **Appendixes**

**Appendix 1: llmservice.py**

import os

import json

from openai import AzureOpenAI

LLM\_ENDPOINT = "https://noland-capstone-ai.openai.azure.com/"

LLM\_DEPLOYMENT = "gpt-4o-mini"

API\_VERSION = "2024-05-01-preview"

def get\_review\_summary(llm\_key,reviews):

"""Initialize the OpenAI client, provide system message, pass json reviews to OpenAI Model"""

reviews\_list=[]

# Iterate over the reviews for each place

for review in reviews:

review\_data = {

"review\_text": review.get("text", {}).get("text", "")

}

reviews\_list.append(review\_data)

# Initialize the Azure OpenAI client...

client = AzureOpenAI(

azure\_endpoint = LLM\_ENDPOINT,

api\_key=llm\_key,

api\_version= API\_VERSION

)

# Create a system message

system\_message = """You are a local travel advisor that summarizes customer reviews.

Summarize the review\_text field in four sentences. The review should be in one paragraph not in bullet format.

"""

messages\_array = [{"role": "system", "content": system\_message}]

messages\_array.append({"role": "user", "content": json.dumps(reviews\_list, indent=2)})

response = client.chat.completions.create(

model=LLM\_DEPLOYMENT,

temperature=0.0,

max\_tokens=1200,

messages=messages\_array

)

generated\_text = response.choices[0].message.content

# Add generated text to message array

messages\_array.append({"role": "assistant", "content": generated\_text})

return generated\_text

**Appendix 2: vlmservice.py**

import os

import aiohttp

import asyncio

import json

import requests

import openai

from fastapi import HTTPException

#Defining config information

VLM\_ENDPOINT = "https://noland-capstone-ai.openai.azure.com/"

VLM\_DEPLOYMENT = "gpt-4"

API\_VERSION = "2024-05-01-preview"

MAX\_CONCURRENT\_REQUESTS = 25

semaphore = asyncio.Semaphore(MAX\_CONCURRENT\_REQUESTS)

def get\_safe\_prompt(keywords,vlm\_key):

HEADERS = {

"Content-Type": "application/json",

"api-key": vlm\_key,

"User-Agent": "Image-Analysis-Tool/1.0"

}

"""Synchronously generate a safe user prompt based on keywords using ChatGPT."""

if not keywords.strip(): # Check for empty or whitespace-only string

return "Describe the objects and setting in the image in a neutral manner."

payload = {

"messages": [

{"role": "system", "content": "You are an AI assistant that reformulates user-provided keywords into safe and neutral image analysis prompts."},

{"role": "user", "content": f"Given the following information: {keywords}, create a professional and neutral prompt that can be used to describe an image focusing on these elements. Do not include any potentially sensitive content."}

],

"model": VLM\_DEPLOYMENT,

"max\_tokens": 100,

"temperature": 0.3

}

response = requests.post(

f"{VLM\_ENDPOINT}/openai/deployments/{VLM\_DEPLOYMENT}/chat/completions?api-version={API\_VERSION}",

headers=HEADERS,

json=payload,

timeout=60

)

if response.status\_code == 200:

data = response.json()

return data.get("choices", [{}])[0].get("message", {}).get("content", "No response content")

elif response.status\_code == 401:

raise Exception(response.text)

else:

return "Describe the objects and setting in the image in a neutral manner."

async def analyze\_image(encoded\_image, safe\_prompt, vlm\_key, retry\_count=0, max\_retries=8):

"""Analyze a binary image using Azure OpenAI GPT-4 Vision."""

HEADERS = {

"Content-Type": "application/json",

"api-key": vlm\_key,

"User-Agent": "Image-Analysis-Tool/1.0"

}

async with aiohttp.ClientSession() as session:

payload = {

"messages": [

{"role": "system", "content": "You are an AI vision model that analyzes images and provides factual descriptions of primary objects, settings, and scenes in four sentences or less, without speculation or interpretation."},

{"role": "user", "content": [

{"type": "text", "text": safe\_prompt},

{"type": "image\_url", "image\_url": f"data:image/jpeg;base64,{encoded\_image}"}

]}

],

"model": VLM\_DEPLOYMENT,

"max\_tokens": 150,

"temperature": 0.3

}

async with session.post(

f"{VLM\_ENDPOINT}/openai/deployments/{VLM\_DEPLOYMENT}/chat/completions?api-version={API\_VERSION}",

headers=HEADERS,

json=payload,

timeout=aiohttp.ClientTimeout(total=120)

) as response:

if response.status == 200:

data = await response.json()

result = data.get("choices", [{}])[0].get("message", {}).get("content", "No response content")

return result

if response.status == 429: # Rate limit error

if retry\_count < max\_retries:

wait\_time = 2 \*\* retry\_count

await asyncio.sleep(wait\_time)

return await analyze\_image(encoded\_image, session, safe\_prompt, retry\_count + 1, max\_retries)

else:

return "Rate limit exceeded after multiple retries"

elif response.status == 400:

response\_text = await response.text()

if "jailbreak" in response\_text.lower() or "content filter" in response\_text.lower():

return "Content blocked due to moderation policy"

elif response.status == 401:

raise HTTPException(status\_code=401,detail="Your VLM key or endpoint is incorrect")

async def generate\_summary(image\_descriptions: list, vlm\_key: str):

"""Generate a summary paragraph based on all image descriptions."""

HEADERS = {

"Content-Type": "application/json",

"api-key": vlm\_key,

"User-Agent": "Image-Analysis-Tool/1.0"

}

if not image\_descriptions:

return "No images were analyzed."

descriptions=[description["vlm\_insight"] for description in image\_descriptions]

# Create a prompt to summarize the descriptions

all\_descriptions = "\n\n".join(descriptions)

num\_descriptions = len(descriptions)

summary\_prompt = f"""Based on the following {num\_descriptions} image descriptions,

create a concise one-paragraph summary that captures the key themes,

settings, and objects that appear across multiple images:

{all\_descriptions}

Summarize these descriptions in one coherent paragraph:"""

payload = {

"messages": [

{"role": "system", "content": "You are an AI assistant that creates concise, informative summaries."},

{"role": "user", "content": summary\_prompt}

],

"model": VLM\_DEPLOYMENT,

"max\_tokens": 250,

"temperature": 0.3

}

async with aiohttp.ClientSession() as session:

async with session.post(

f"{VLM\_ENDPOINT}/openai/deployments/{VLM\_DEPLOYMENT}/chat/completions?api-version={API\_VERSION}",

headers=HEADERS,

json=payload,

timeout=aiohttp.ClientTimeout(total=120)

) as response:

if response.status == 200:

data = await response.json()

return data.get("choices", [{}])[0].get("message", {}).get("content", "Failed to generate summary.")

else:

return f"{response.status} something went wrong"

**Appendix 3: Standalone / local hosted version setup steps**

Download Docker Desktop:

* Download Docker Desktop to your local computer using the following link:
* <https://www.docker.com/products/docker-desktop/>

Login to Docker Desktop:

* Login to Docker Desktop using your google email account. In Docker Desktop go to the Images tab and open the terminal window.
* If you are running the codebase in WSL and running Docker Desktop on Windows ensure they are connected: Open Docker Desktop → Settings → Resources → WSL Integration

Build and Run Images:

* Clone the github repo or download code zipfile.
* Using VSCODE and the integrated terminal navigate to the application parent directory "cd capstoneAI2C" or "cd capstoneAI2C-main"
* Execute the below command in the VSCODE integrated terminal to use docker-compose to build the frontend and backend Docker Images using the docker-compose.yaml and run the images
* docker-compose up -d --build
* The above command will take approximately 90 seconds to complete. The terminal output from this command will look like the below:

✔ backend Built

✔ frontend Built

✔ Network capstoneai2c\_default Created

✔ Container backend Started

✔ Container frontend Started

* To verify the containers are running, execute the following command in the VSCODE integrated terminal. Ensure the status is listed as "Up".
* docker ps

Access the Application:

* To access the application open a browser and paste in the below URL:
* http://localhost/

Stopping the Application:

* In the event there is a need stop the containers run the following command:
* docker-compose down --volumes
* If the output of this command is “no such service -volumes”, ensure there are two dashes (-) before volumes. The correct output from this command is, "container frontend removed, container backend removed, and network my-docker-project\_default removed” each printed on their own separate line.
* To ensure there are no containers running execute the following:
* docker ps

Admin Note: The docker engine needs to run to allow docker compose to run the containers. To ensure this occurs go to Docker Desktop and click on the settings gear in the upper right hand corner. Under the General tab ensure the box labeled “Start Docker Desktop when you sign in to your computer” is checked.

**Appendix 4: Application prerequisites and dependencies**

* Prerequisites
  + Python 3.9+
  + VSCODE
  + Google Maps API Key
  + OpenAI API Key
* Backend requirements.txt file:
  + googlemaps
  + uvicorn
  + requests
  + fastapi
  + httpx
  + azure-data-tables
  + pytest-asyncio
  + openai
  + aiohttp
  + googletrans
  + Deep\_translator
* Frontend requirements.txt file:
  + streamlit==1.41.1
  + XlsxWriter==3.2.2
  + simplekml==1.3.6
  + requests==2.32.3
  + folium==0.19.5
  + branca==0.8.1
  + streamlit\_folium==0.24.0

## **References**

* Microsoft. (n.d.). *Azure OpenAI Service models*. Microsoft Learn. Retrieved February 18, 2025, from<https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models?tabs=global-standard%2Cstandard-chat-completions#gpt-4o-and-gpt-4-turbo>
* Microsoft. (n.d.). *Azure OpenAI Service models.* Retrieved February 18, 2025, from<https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models>
* OpenAI. (n.d.). *GPT-4 Turbo Overview.* Retrieved February 18, 2025, from<https://openai.com/research/gpt-4-turbo>
* OpenAI. (n.d.). Introduction to GPT-4o. OpenAI Cookbook. Retrieved February 18, 2025, from<https://cookbook.openai.com/examples/gpt4o/introduction_to_gpt4o>
* Microsoft, "Azure Kubernetes Service (AKS)," *Microsoft Azure*, [Online]. Available:<https://azure.microsoft.com/en-us/products/kubernetes-service>.