**A robot in a lab

Description automatically generated**

**AI Alchemists – Hackathon 2024**

**Documentation**

Team Members:

**Raluca-Maria Bacioiu**

**Mario Dumitru**

**Andreea-Carmen Lemnaru**

**Bogdan Peltea**

**Irina-Florina Zaharescu**

A black text on a white background

Description automatically generated

Contents

[1 Introduction 3](#_Toc166276308)

[Objectives 3](#_Toc166276309)

[2 User Interface Design 4](#_Toc166276310)

[2.1 Web App – User Journey 4](#_Toc166276311)

[2.2 Summary Reports 6](#_Toc166276312)

[2.3 Error Handling: 8](#_Toc166276313)

[3 Application Architecture and Workflow 9](#_Toc166276314)

[4 Technology Stack 13](#_Toc166276315)

[Microservices Architecture 13](#_Toc166276316)

[Single Responsibility Principle 14](#_Toc166276317)

[CI/CD 15](#_Toc166276318)

[Microservices Overview 16](#_Toc166276319)

[Cloud Run 16](#_Toc166276320)

[Eventarc 17](#_Toc166276321)

[Cloud Functions 17](#_Toc166276322)

[BigQuery 17](#_Toc166276323)

[Flask 18](#_Toc166276324)

[Firestore 18](#_Toc166276325)

[5 Features & Functionalities 18](#_Toc166276326)

[5.1 Features 18](#_Toc166276327)

[5.2 Functionalities 18](#_Toc166276328)

[Format\_classifier 18](#_Toc166276329)

[Image\_handler 19](#_Toc166276330)

[Document\_handler 20](#_Toc166276331)

[IoT\_handler 21](#_Toc166276332)

[Video\_handler 21](#_Toc166276333)

[Text\_processor 22](#_Toc166276334)

[Stats\_generator 22](#_Toc166276335)

[6 Backend of the frontend 22](#_Toc166276336)

[7 Limitations and Future Improvements 25](#_Toc166276337)

[Limitations 25](#_Toc166276338)

[Improvements 25](#_Toc166276339)

# Introduction

In our contemporary society, the spread of unverified and fake information has become an omnipresent challenge, casting a shadow over various aspects of our lives. Nowhere is this more evident and concerning than in the field of healthcare, where the dissemination of misinformation can have dire consequences for individuals' health and well-being.

In an age dominated by social media platforms and instant communication channels, misinformation spreads like wildfire, often masquerading as legitimate advice or scientific fact. From dubious health remedies touted as miraculous cures to conspiracy theories undermining the efficacy of proven medical interventions, the sheer volume and accessibility of such misinformation pose a significant threat to public health.

Moreover, the impact of misinformation extends beyond individual health outcomes, permeating societal attitudes and behaviors towards public health measures such as vaccinations, disease prevention, and healthcare policies. In an era where the global community faces unprecedented health challenges, the proliferation of misinformation only serves to exacerbate these crises, hindering efforts to contain outbreaks and safeguard public health.

Robust fact-checking mechanisms, increased media literacy initiatives, and transparent communication channels are essential tools in combating the spread of misinformation and fostering a culture of evidence-based decision-making.

Taking all of this into consideration, we developed an application driven by the goal of leveraging technology to enhance access to verified healthcare studies and address the information needs of individuals seeking medical assistance and information.

This is a solution that uses a GEN AI foundational model (GPT-4) to analyze the content of various marketing materials, identify & categorize facts/claims about the product, and reference them to authorized sources such as research papers, clinical trials, and relevant medical studies.

Generative Pre-trained Transformer 4 (GPT-4) is a multimodal large language model created by OpenAI, and the fourth in its series of GPT foundation models. It was launched on March 14, 2023, and made publicly available via the paid chatbot product ChatGPT Plus, via OpenAI's API, and via the free chatbot Microsoft Copilot. As a transformer-based model, GPT-4 uses a paradigm where pre-training using both public data and "data licensed from third-party providers" is used to predict the next token. After this step, the model was then fine-tuned with reinforcement learning feedback from humans and AI for human alignment and policy compliance.

GPT-4 is not only more powerful than GPT-3.5, but it's also multimodal, meaning it's capable of analyzing text, images, and voice.

This advanced version was trained on information as recent as April 2023, and for our solution this was enhanced by the integration with PubMed API, the world's largest database for biomedical and health sciences literature.

PubMed is a free resource supporting the search and retrieval of biomedical and life sciences literature with the aim of improving health–both globally and personally.

The PubMed database contains more than 37 million citations and abstracts of biomedical literature. It does not include full text journal articles; however, links to the full text are often present when available from other sources, such as the publisher's website or PubMed Central (PMC). Citations in PubMed primarily stem from the biomedicine and health fields, and related disciplines such as life sciences, behavioral sciences, chemical sciences, and bioengineering.

## Objectives

Create a proof of concept (POC) employing GenAI models to analyze the content of various marketing materials, identify & categorize facts/claims about the product, and reference them to authorized sources such as research papers, clinical trials, and relevant medical studies, in order to streamline the medical-legal review process.

This involves assessing the precision of medical data across diverse marketing materials, verifying whether assertions regarding a drug are substantiated or refuted by scientific data, and ensuring transparent and efficient communication of the drug's associated risks.

A medical-legal review (MLR) is an essential process in healthcare marketing to make sure advertising and promotional content is accurate, compliant, and ready to be published. This all the content (from illustrations to animations, and text) undergoes before publication. MLR process will help companies to avoid risks related to patients’ health, legal actions, financial loss due to fines or compensations, brand reputation, etc.

# User Interface Design

## Web App – User Journey

Link: [Login (app-adwexujega-ey.a.run.app)](https://app-adwexujega-ey.a.run.app/login)

From the first screen of the app the user can either register or login into an existing account.

**Step 1:**

A new user should register and create an account in order to be able to access the app.

A screenshot of a login form

Description automatically generated A screenshot of a login form

Description automatically generated

The data is automatically encrypted, physical passwords are not stored, and everything is stored in a BigQuery table.

**Step 2:**

The user should use the credentials to log in.

A screenshot of a login form

Description automatically generated

**Step 3:**

After logging in, the next screen is the main screen with the “Document Upload” feature and the “Chat”, where you can start the conversation with the AI engine.

One interesting feature is that the app stores the history of search for each unique user and this is not accessible by other user.

A screenshot of a chat

Description automatically generated

There can be multiple scenarios that can be accessed from the interface:

1. Ask the AI Engine a question
2. Ask the AI Engine a question along with a document file upload
3. Ask the AI Engine a question along with a picture upload
4. Ask the AI Engine a question along with a audio/video upload
5. Ask the AI Engine a question along with a website link/youtube link

A screenshot of a chat

Description automatically generated

**Step 4:**

The user can start a new Conversation with the AI Engine, if he doesn’t want to keep the history of asked questions.

A screenshot of a computer

Description automatically generated

## 2.2 Summary Reports

An admin user, which is a normal registered user that has the admin flag manually set to “true” in the BQ user table, was created to have access to “Expose Statistics” button. This button will trigger a graph with the most frequently encountered words in conversations between the chat and all the users.

A screenshot of a phone

Description automatically generated A screenshot of a computer

Description automatically generated

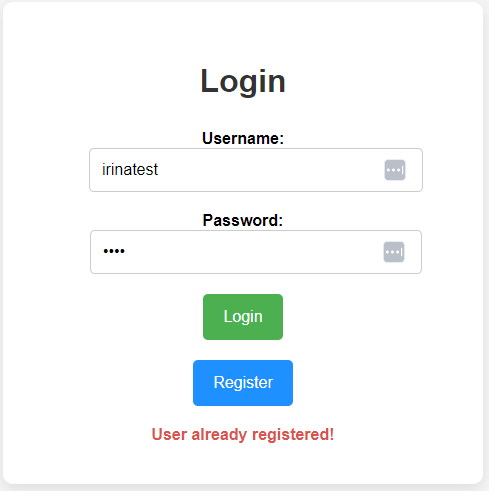
The admin user can select from the Statistics page the period he wants to generate the graph.

A screenshot of a graph

Description automatically generated

## Error Handling:

1. Already registered user



1. Incorrect username or password

A screenshot of a login screen

Description automatically generated

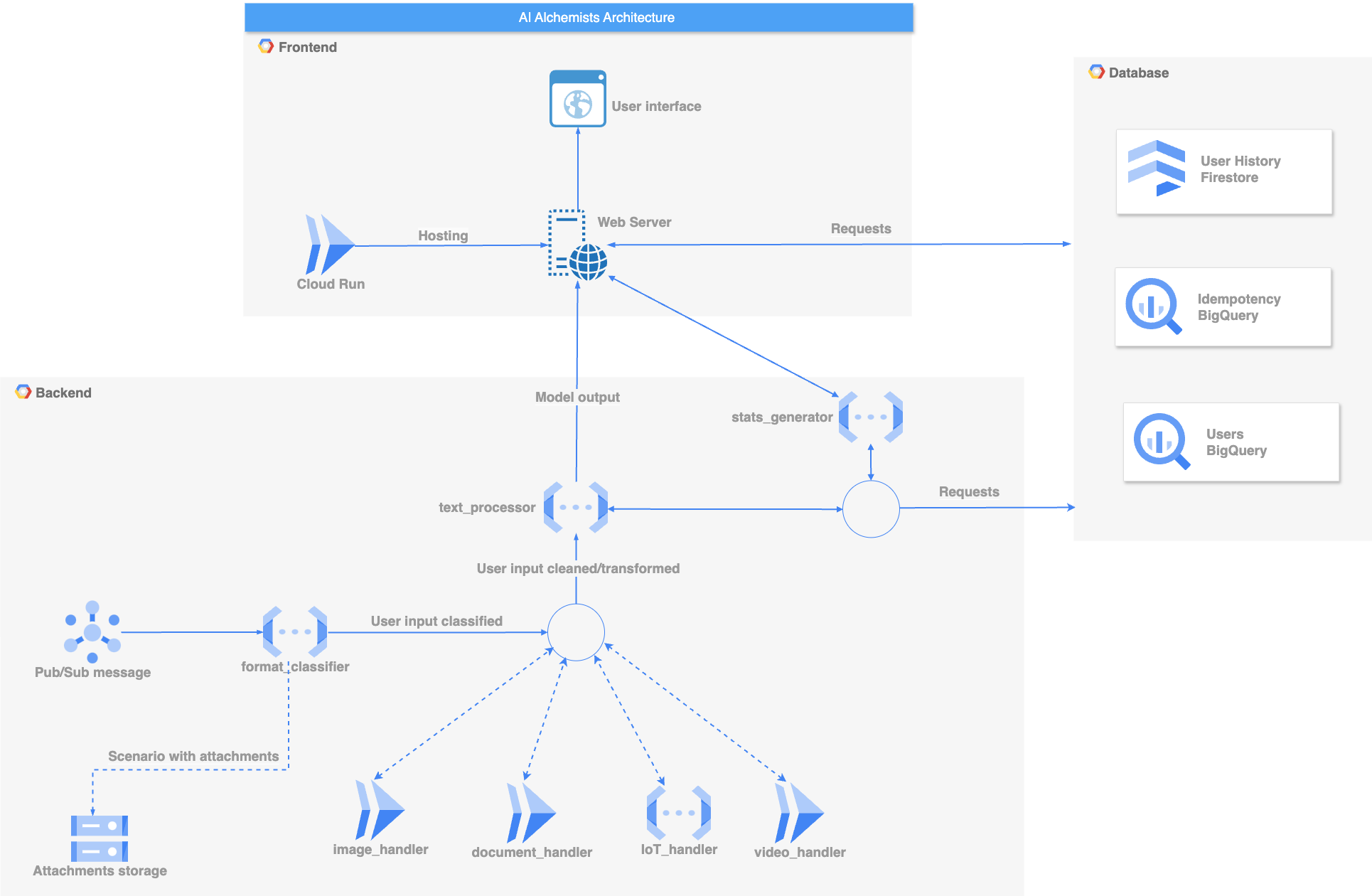
1. No question addressed to the AI Engine

The AI Engine was developed to respond only if it received a question or request.

A screenshot of a chat

Description automatically generated

# Application Architecture and Workflow



The pipeline starts when the user addresses a question in the interface, following one of the above mentioned scenarios. This action will send a message to format\_classifer cloud function and the function will validate the message according to the list of supported extensions:

* video\_file\_extensions =["wav", "mp3", "mp4", "mpeg", "mpga", "m4a", "webm"]
* image\_file\_extensions =["jpg", "jpeg", "png", "gif", "bmp", "tiff", "tif", "raw", "svg", "webp"]
* document\_file\_extensions = ["doc", "docx", "xls", "xlsx", "ppt", "pptx", "pdf", "txt", "rtf", "csv", "odt", "ods", "odp", "pages", "numbers", "key"]

The files uploaded by the user in the interface are stored in a dedicated Bucket (alchemist\_ingestion\_data\_707f0d). The Bucket was created with a retention policy of one day.

Moreover, if the user provides any link in the question/request, the cloud function filters the links using regex patterns:

* video\_link\_pattern=re.compile(r"https?:\/\/(?:www\.)?(?:youtube\.com\/(?:[^\/\n\s]+\/\S+\/|(?:v|e(?:mbed)?)\/|\S\*?[?&]v=)|youtu\.be\/)\S+")
* webpage\_link\_pattern = re.compile(r"https?:\/\/[^\s]+")

If no attachment has been identified, it means that the user addressed a simple question. This is ensured by 2.3.c chapter (error handling).

**Attachment Maneuvering**

After the files/links are classified, the corresponding function is triggered through Pub/Sub.

**Images**

For images it is used the image\_handler function, which takes the file from the previous mentioned bucket, generates a comprehensive description of the parsed picture, using vikhyatk/moondream2 Hugging Face model, version of 2024-04-02. In this way, we ensure that not only images containing text are processed. The model is able to extract context even from non-textual images. Meaning that it is able to identify diseases, conditions and medications to say the least.

The function is deployed using Cloud Run and in order to prevent multiple processing of the same input, a check of the event id is done. If the event id is already present in the specific Big Query table (idempotency.image\_handler\_msg\_ids), the function is not triggered. Otherwise, the event id is inserted in the table and the function starts execution.

Further, a Pub/Sub message which contains the extracted output is sent to the next function in the pipeline, text\_processor.

**Documents**

For documents it is used the document\_handler function, which takes the file from the previous mentioned bucket, extracts the content of the document using tika package.

The function is deployed using Cloud Run and in order to prevent multiple processing of the same input, a check of the event id is done. If the event id is already present in the specific Big Query table (idempotency.document\_handler\_msg\_ids), the function is not triggered. Otherwise, the event id is inserted in the table and the function starts execution.

Further, a Pub/Sub message which contains the extracted output is sent to the next function in the pipeline, text\_processor.

**Audio/Video**

For audio/video it is used the video\_handler function, which takes the file from the previous mentioned bucket, transcribes the audio from the file using whisper package. If a Youtube link was inserted in the user request, the function first has to download the video file and store it in a temporary location and then parses it to the whisper model.

The function is deployed using Cloud Run and in order to prevent multiple processing of the same input, a check of the event id is done. If the event id is already present in the specific Big Query table (idempotency.video\_handler\_msg\_ids), the function is not triggered. Otherwise, the event id is inserted in the table and the function starts execution.

Further, a Pub/Sub message which contains the extracted output is sent to the next function in the pipeline, text\_processor.

**Web Pages**

For web pages it is used the IoT\_handler function, which extracts the link from the Pub/Sub message

It retrieves the URL from the event data, makes a request to the web page, parses the HTML content using BeautifulSoup package.

Further, a Pub/Sub message which contains the extracted output is sent to the next function in the pipeline, text\_processor.

**Model Invocation**

This part is ensured by text\_processor. If the user asks only a question with no attachment, the model will try to respond to that specific question without additional context. If the user asks a question along with an attachment, the model will try to respond taking into account the context offered by the attachment.

AI model used is gpt-4 provided by AzureChatOpenAI, version as of 2024-02-01. The solution uses a predefined LangChain Tool Calling agent(create\_tool\_calling\_agent), which allows a model to detect when one or more tools should be called and respond with the inputs that should be passed to those tools. The tool used for extending the medical knowledge is PubMed.

Its functionality is shown below in the comparison GPT-3.5 model vs the AI Engine:

A white background with black text

Description automatically generated

A screenshot of a medical document

Description automatically generated

Domain restriction was done by using prompts.

The prompt used for processing the question along with the attachment output:

A black background with text

Description automatically generated

Prompt template used for processing the question alone:

A black background with orange and white lines

Description automatically generated

The model keeps tracks of the conversation history by querying the Firestore database which stores the user's input from the interface and the model's output.

An additional feature is available only for the admin users which allows them to expose statistics of the most frequent words identified in the overall conversation of the model with all the users. The user has an option to select from the interface the timeframe he wants to select: today, 7 days, last month, last 6 months, last year

In the background, this functionality calls stats\_generator function which extracts the entire conversation history of the model and gives it as input to a LLM model (ConversationChain provided by AzureChatOpenAI) which identifies the most common medical words used and calculates their frequency.

# Technology Stack

## Microservices Architecture

The solution operates entirely on Cloud and was developed on a microservices architecture.

Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications. Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility. To serve a single user request, a microservices-based application can call on many internal microservices to compose its response.

A microservices architecture is a type of application architecture where the application is developed as a collection of services. It provides the framework to develop, deploy, and maintain microservices architecture diagrams and services independently. Within a microservices architecture, each microservice is a single service built to accommodate an application feature and handle discrete tasks. Each microservice communicates with other services through simple interfaces to solve business problems.

Typically, microservices are used to speed up application development.

When you use Google Cloud, you can easily deploy microservices using either the managed container service, Google Kubernetes Engine, or the fully managed serverless offering, Cloud Run.

Main advantages of microservices architecture:

* Scalability improvements

Since each microservice runs independently, it is easier to add, remove, update or scale each cloud microservice. Developers can perform these tasks without disrupting any other microservice in the system.

* Improved fault isolation

Under a monolithic architecture structure, when developers experience a failure in one element of the architecture, it will collapse all architecture components. With a microservices architecture, if one service fails, it’s much less likely that other parts of the application will fail because each microservice runs independently.

* Program language and technology agnostic

A microservice application can be programmed in any language, so dev teams can choose the best language for the job. Using cloud-based microservices gives developers another advantage, as they can access an application from any internet-connected device, regardless of its platform.

* Simpler to deploy

A microservices architecture lets teams deploy independent applications without affecting other services in the architecture. This feature, one of the pros of microservices, will enable developers to add new modules without redesigning the system's complete structure.

* Reusability across different areas of business

Some microservice applications may be shareable across a business. If a site has several different areas, each with a login or payment option, the same microservice application can be used in each instance.

* Faster time-to-market

Developers can plug this new “microsurgery” into the architecture without fear of conflicts with other code or of creating service outages that ripple across the website. Development teams working on different microservices don't have to wait for each other to finish.

* Ability to experiment

It’s simple to roll out new features because each service is independent of the others. If customers don't like it, or the business benefits aren’t clear, it's much easier to roll it back without affecting the rest of the operation.

* Improved data security

If the components of the computer systems architecture break down into smaller pieces, sensitive data is protected from intrusions from another area. While there are connections between all microservices, developers can use secure APIs to connect the services. Secure APIs safeguard data by ensuring it is only available to specifically authorized users, applications and servers.

* Outsourcing flexibility

It may be necessary for a business to outsource certain functions to third-party partners. Many companies are concerned about protecting intellectual property with a monolithic architecture format. However, a microservices architecture allows businesses to segment areas just for partners that won’t otherwise disclose core services.

## Single Responsibility Principle

The solution was developed with adherence to the Single Responsibility Principle.

The Single Responsibility Principle (SRP) is a computer programming principle that states that "A module should be responsible to one, and only one, actor." The term actor refers to a group (consisting of one or more stakeholders or users) that requires a change in the module. SRP refers to one of the SOLID principles, which states that a class should have only one reason to change, meaning it should have only one responsibility.

In practical terms, this means that each class should encapsulate only one aspect of the program's functionality or represent only one concept. By adhering to the SRP, code becomes more modular, easier to understand, and less prone to bugs. It also promotes better code organization and facilitates easier maintenance and changes in the future.

## CI/CD

CI/CD stands for Continuous Integration and Continuous Delivery (or Continuous Deployment). It's a set of best practices and methodologies in software development aimed at improving the efficiency, quality, and speed of delivering software changes.

Terraform is an open-source infrastructure as code (IaC) tool developed by HashiCorp. It allows users to define and provision infrastructure resources using a declarative configuration language. With Terraform, you can manage infrastructure across various cloud providers (such as AWS, Azure, Google Cloud Platform), as well as on-premises environments and third-party services.

Overall, Terraform simplifies and automates the process of provisioning, configuring, and managing infrastructure resources, enabling infrastructure as code practices and improving the reliability, scalability, and efficiency of infrastructure management.

**CI/CD with Terraform** involves integrating Terraform, an infrastructure as code (IaC) tool, into a Continuous Integration/Continuous Deployment (CI/CD) pipeline to automate the provisioning and management of cloud infrastructure.

Here's how it typically works:

1. **Continuous Integration (CI):**

* Developers push code changes to a version control system (e.g., Git).
* A CI server (e.g., Jenkins, CircleCI, GitLab CI) monitors the repository for changes.
* When changes are detected, the CI server triggers a build process.
* During the build process, Terraform code is validated and tested alongside application code.
* Unit tests for Terraform configurations may be executed to ensure the correctness of infrastructure code.

1. **Continuous Deployment (CD):**

* After successful CI, the CD process begins.
* Terraform configurations are applied to create, update, or delete cloud resources based on the changes in the Terraform code.
* Terraform applies changes in a controlled and automated manner, ensuring that infrastructure remains consistent and in the desired state.
* CD pipelines may include multiple stages (e.g., development, staging, production), each with its own set of Terraform configurations and deployment strategies.

1. **Infrastructure as Code (IaC) Best Practices:**

* Terraform configurations are stored alongside application code in the same version control system, ensuring that infrastructure changes are tracked and versioned.
* Terraform code is modular and reusable, promoting consistency and reducing duplication.
* Infrastructure changes are reviewed through code review processes similar to application code.
* Terraform state files are managed centrally (e.g., using remote state storage) to enable collaboration and ensure consistency across environments.

By incorporating Terraform into CI/CD pipelines, teams can automate the management of infrastructure changes, improve consistency and reliability, and accelerate the delivery of applications to production environments.

## Microservices Overview

### Cloud Run

Cloud Run is a fully managed platform that enables you to run your code directly on top of Google’s scalable infrastructure. Cloud Run is simple, automated, and designed to make you more productive.

You can deploy code written in any programming language on Cloud Run if you can build a container image from it. In fact, building container images is optional. If you're using Go, Node.js, Python, Java, .NET Core, or Ruby, you can use the source-based deployment option that builds the container for you, using the best practices for the language you're using.

Google has built Cloud Run to work well together with other services on Google Cloud, so you can build full-featured applications.

In short, Cloud Run allows developers to spend their time writing their code, and very little time operating, configuring, and scaling their Cloud Run service. You don't have to create a cluster or manage infrastructure in order to be productive with Cloud Run.

On Cloud Run, your code can either run continuously as a service or as a job. Both services and jobs run in the same environment and can use the same integrations with other services on Google Cloud.

Cloud Run services. Used to run code that responds to web requests, or events.

Cloud Run jobs. Used to run code that performs work (a job) and quits when the work is done.

Cloud Run adds and removes instances automatically to handle all incoming requests or to handle increased CPU utilization outside requests if CPU allocation is set to always on. If there are no incoming requests to your service, even the last remaining instance will be removed. This behavior is commonly referred to as scale to zero.

If there are no active instances, a new instance is created on-demand as soon as a request comes in. This negatively impacts the response time for these first requests, depending on how fast your container becomes ready to handle requests.

To make sure your service doesn't scale to zero instances, you can configure Cloud Run to keep a minimum amount of instances active.

### Eventarc

**Eventarc** is a service provided by Google Cloud Platform (GCP) that enables event-driven architectures by allowing you to consume events from Google Cloud services and third-party sources. It provides a unified eventing experience across Google Cloud, allowing you to easily trigger serverless functions, workflows, or other event-driven applications in response to events.

With Eventarc, you can subscribe to events from various GCP services such as Pub/Sub, Cloud Storage, Cloud Firestore, and more. Additionally, it supports external sources like GitHub repositories and webhooks.

Eventarc simplifies the process of building event-driven applications by providing a consistent way to handle and respond to events across different services and sources within the Google Cloud ecosystem.

**Eventarc** makes it easy to connect various services (Cloud Run, Cloud Functions, Workfklows) with events from a variety of sources. It allows you to build event-driven architectures in which microservices are loosely coupled and distributed. It also takes care of event ingestion, delivery, security, authorization, and error-handling for you which improves developer agility and application resilience.

### Cloud Functions

Cloud Functions is a serverless compute service provided by Google Cloud Platform (GCP). It allows you to run event-driven code in a fully managed environment without the need to provision or manage servers.

With Cloud Functions, you can write small, single-purpose functions that automatically respond to events from various Google Cloud services or HTTP requests. These functions are triggered by events such as changes in Cloud Storage, messages in Pub/Sub, HTTP requests to a specified endpoint, or changes in Firestore database.

Cloud Functions are typically used for tasks such as data processing, file manipulation, API integrations, or backend logic for web and mobile applications. They are designed to scale automatically in response to load, ensuring that your functions can handle spikes in traffic without manual intervention.

Overall, Cloud Functions provide a convenient and cost-effective way to run code in response to events, allowing developers to focus on writing code rather than managing infrastructure.

### BigQuery

BigQuery tables offer several advantages: scalability, managed service, serverless, performance, integration, security. Overall, BigQuery tables offer a powerful and flexible platform for storing, analyzing, and visualizing large-scale datasets, making it an excellent choice for many data warehousing and analytics use cases.

### Flask

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.

### Firestore

Firestore is a flexible, scalable, and fully managed NoSQL document database provided by Google Cloud Platform (GCP). It's part of the Firebase platform, which offers a suite of tools for building mobile and web applications.

Firestore is commonly used in a wide range of applications, including mobile and web apps, IoT devices, and backend services. It provides a flexible and scalable data storage solution with real-time synchronization and offline support, making it suitable for building modern, responsive, and scalable applications.

# Features & Functionalities

## 5.1 Features

Some of the solution’s features:

* **Upload documents**: Multiple types of documents, including video, images, and text can be uploaded through the solution.
* **Domain restriction**: The solution uses LLM to restrict the conversation to the medical domain and sends a message to inform the user of the restriction.
* **Chat history**: The application allows you to keep the chat history and continue the conversation or initiate a new conversation.
* **Statistics**: The admin user is able to see statistics generated based on conversation history data between users and an AI model. It interacts with Firestore, BigQuery, and external APIs to extract, classify and analyze conversation data.

## 5.2 Functionalities

### Format\_classifier

This cloud function receives Pub/Sub messages, verifies user’s input, classifies it and triggers the respective function: image\_handler, video\_handler, doc\_handler or IoT\_handler.

It is considered the link between front-end and back-end, the first bridge of connection.

**Functions overview:**

* **detect\_file\_type(filename)**: Determines the type of file based on its extension. It checks the file extension against predefined lists of known extensions for video, image, and document files. If the extension matches, it returns the corresponding trigger for handling the file type.
* **detect\_text\_type(text)**: Detects the type of text content and extracts URLs if present. It uses regular expressions to identify patterns such as URLs, video links, webpage links, and plain text. Based on the detected patterns, it returns the appropriate trigger for handling the text type along with any detected URLs.
* **format\_classifier(event, context)**: Formats and classifies Pub/Sub messages. It decodes the message, extracts relevant information such as the statement and file path, and then determines the appropriate trigger for handling the message based on its content. It calls the detect\_file\_type() or detect\_text\_type() function as needed and forwards the message to the corresponding processing component.

**Key Concepts:**

* Detection of file types based on file extensions.
* Detection of text types including URLs, plain text, and video links.
* Publishing messages to corresponding triggers based on detected types.

This script is particularly useful for a system handling various types of content, allowing it to route messages to the appropriate processing functions based on their content types.

### Image\_handler

This cloud function receives Pub/Sub messages, processes image data contained in the messages, extracts textual content from the images, and publishes new messages containing the extracted text. It leverages Google Cloud services like Cloud Storage, BigQuery, and Firestore for data storage and processing.

1. **Functions Overview:**

* **check\_events\_duplicates(event\_id):** Checks if an event ID already exists in a BigQuery table to prevent duplicate processing. It queries a BigQuery table to check for the existence of the event ID and inserts the event ID into the table if it doesn't already exist.
* **check\_firestore\_state(uuid):** Checks the state of a Firestore document to determine the type of content it holds. It queries a Firestore collection for documents with a specified UUID and returns the type of content stored in the latest document.
* **extract\_content\_from\_image(image\_path):** Extracts textual content from an image using a pre-trained model. It loads the image from the specified path, encodes the image, and generates textual content by querying the model.
* **image\_handler(pubsub\_message):** Processes images received via Pub/Sub messages, extracts textual content from the images using the extract\_content\_from\_image function, cleans the extracted text, and publishes a new message containing the original statement, the extracted text, and the UUID of the image.

1. **Flask App Route:**

* Defines a route for handling POST requests at the root URL ("/"). It receives Pub/Sub messages, checks their validity, and processes them if they contain image data.
* Checks if the received message is in the expected format, decodes the message if it contains base64 encoded data, extracts the message ID as context, checks for duplicate events using check\_events\_duplicates, and verifies the state of the Firestore document using check\_firestore\_state.
* If the message is valid and not a duplicate event, and the Firestore document contains a statement, it calls the image\_handler function to process the image and extract textual content.
* Returns a HTTP response with a status code 200 indicating successful processing.

1. **Main Execution:**

* Runs the Flask application if it's executed directly.

### Document\_handler

This cloud function receives Pub/Sub messages, processes document data contained in the messages, extracts textual content from the documents using the Tika library, and publishes new messages containing the extracted text. It leverages Google Cloud services like Cloud Storage and BigQuery for data storage and processing.

**1. Functions Overview:**

* **check\_events\_duplicates(event\_id):** This function checks if an event ID already exists in a BigQuery table to prevent duplicate processing. It queries a BigQuery table to check for the existence of the event ID and inserts the event ID into the table if it doesn't already exist.
* **extract\_text\_from\_doc(file\_path):** This function extracts text from a document using the Tika library. It takes the path to the document as input and returns the extracted text.
* **document\_handler(pubsub\_message):** This function processes documents received via Pub/Sub messages. It extracts textual content from the documents using the extract\_text\_from\_doc function, cleans the extracted text, and publishes a new message containing the original statement, the extracted text, and the UUID of the document.

1. **Flask App Route:**

* Defines a route for handling POST requests at the root URL ("/"). This route is intended to be the entry point for Pub/Sub messages.
* Checks if the received message is in the expected format, decodes the message if it contains base64 encoded data, extracts the message ID as context, and checks for duplicate events using check\_events\_duplicates.
* If the message is valid and not a duplicate event, it calls the document\_handler function to process the document and extract textual content.
* Returns a HTTP response with a status code 200 indicating successful processing.

1. **Main Execution:**

* Runs the Flask application if it's executed directly.

### IoT\_handler

This cloud function receives Pub/Sub messages, processes web pages specified in the event data, extracts textual content and sends the processed data via Pub/Sub to another component for further handling.

1. **Functions Overview:**

* **IoT\_handler(cloud\_event):** The function is triggered by a Pub/Sub message. It decodes the message data from the cloud event and retrieves the URL of the web page to be processed. It makes a request to the specified URL to fetch the HTML content of the web page. It parses the HTML content using BeautifulSoup (bs4) to extract the text content of the web page. It formats the extracted content along with other relevant data (statement and UUID) into a JSON-formatted string. It publishes the formatted message to another component using the publish\_message function.

### Video\_handler

This cloud function is designed to handle Pub/Sub messages, specifically for processing video/audio files or YouTube links. It extracts audio into text and publishes the processed data. It leverages Google Cloud services like Cloud Storage and BigQuery for data storage and processing.

1. **Functions Overview:**

* **check\_events\_duplicates(event\_id):** Checks if an event ID already exists in a BigQuery table to prevent duplicate processing. It queries a BigQuery table to check for the existence of the event ID and inserts the event ID into the table if it doesn't already exist.
* **video\_handler(pubsub\_message):** Extracts audio into text from a video file or a YouTube link. It loads the file from the specified path or downloads it if a YouTube link was provided and transcribes the audio by querying the whisper model. It publishes the processed data to another component using the publish\_message function.

1. **Flask App Route:**

Defines a route for handling POST requests at the root URL ("/"). It receives Pub/Sub messages, checks their validity, and processes them if they contain video/audio data or YouTube links.

Checks if the received message is in the expected format, decodes the message if it contains base64 encoded data, extracts the message ID as context, checks for duplicate events using check\_events\_duplicates.

If the message is valid and not a duplicate event, it calls the video\_handler function to process the file and extract textual content.

Returns a HTTP response with a status code 200 indicating successful processing.

1. **Main Execution:**

Runs the Flask application if it's executed directly.

### Text\_processor

The cloud function was designed to process a user's question alongside any files uploaded or links provided by the user. It utilizes an create\_tool\_calling\_agent LLM agent, Pubmed tool, and external services to generate a response to the user's query.

1. **Functions Overview:**

* **extract\_all\_documents\_from\_collection(project\_id, database\_id, collection\_name)**: Extracts all documents from a Firestore collection in a specific database. The documents are represented by the questions addressed by the user in the interface and the answers from the LLM model.
* **remove\_urls(text):** Removes URLs from a given text.
* **text\_processor(cloud\_event):** A cloud function that processes a user's question along with any files uploaded or links provided. It extracts the statement, UUID, and attachment output (if any) from the cloud event data. It queries a BigQuery table to check if the UUID exists. It constructs a prompt template based on the presence of attachment output. It creates an agent using the AzureChatOpenAI model and invokes it with the user's question. It sends the generated response to the interface via a POST request.

### Stats\_generator

This module contains functions to generate statistics based on conversation history data between users and an AI model. It interacts with Firestore, BigQuery, and external APIs to extract, classify, and analyze conversation data.

1. **Functions Overview:**

* **check\_firestore\_documents():** Extracts documents from the history collection in Firestore. Returns a list of documents containing conversation history data.
* **classify\_collection(user\_input):** Classifies documents based on the given timeframe. The timeframe for classifying documents ('today', '7days', 'month', '6months', 'year') is selected by the user in the UI.
* **check\_admin\_users(uuid):** Checks if the user is an admin by querying users BigQuery table
* **stats\_generator(event, context):** Generates statistics based on the provided timeframe. It takes the user's input representing the timeframe selected from the UI and calls the ConversationChain LLM model to extract some medical keywords from the conversation history along with their frequency. Returns a JSON-formatted string containing the keywords and their frequency.

# Backend of the frontend

“Backend of the frontend” is a term used to refer to the server-side components that support a frontend application. It can include services responsible for handling data processing, authentication, authorization, and other backend functionalities that support the frontend user interface. This architecture is often used in modern web applications where there's a clear separation between the frontend and backend components for better maintainability and scalability.

The Node.js application is designed to provide real-time communication, user authentication, file upload handling, and database interaction. The application leverages Express.js for web server functionality, Socket.io for real-time event-based communication, and Multer for handling file uploads.

Below is a detailed breakdown of the application's components and functionalities:

**Dependencies**

* multer: Middleware for handling multipart/form-data, which is commonly used for file uploads.
* express: A minimalist web framework for Node.js, providing robust features for web and mobile applications.
* express-session: Middleware for managing user sessions, enabling the application to maintain user state across multiple HTTP requests.
* socket.io: A library that enables real-time, bidirectional, and event-based communication between web clients and servers.
* queryBigquery: A custom function for querying a BigQuery database.
* registerDbUser: A custom function for registering a user in a database.
* cleanUserConversation: A custom function for cleaning up a user's conversation history.
* chatHistory: A custom function for retrieving chat history from a database.
* updateDb: A custom function for updating records in a database.
* publishMessage: A custom function for publishing messages to a message broker.
* uploadFileToBucket: A custom function for uploading files to a cloud storage bucket.

**Express Setup**

* Initialization: The Express application is initialized using express().
* Middleware Configuration: Middleware functions for parsing JSON requests and serving static files are configured using app.use().
* Session Configuration: Session middleware is configured using express-session to manage user authentication and maintain session state.

**Socket.io Setup**

* Initialization: Socket.io is initialized and configured to listen for WebSocket connections on the Express server.
* Connection Event: An event handler is set up to log when clients connect to the server.

**Authentication Middleware**

* isAuthenticated Middleware: This middleware function checks if a user is authenticated by verifying the presence of a user session. If the user is not authenticated, it redirects the user to the login page.

**Routes**

* Login Routes: Routes for rendering the login page (/login) and handling user login (POST /login).
* Statistics Routes: Routes for rendering the statistics page (/statistics) and handling requests to generate statistics (POST /generatestats).
* Conversation Routes: Route for starting a new conversation (GET /newconversation).
* Registration Route: Route for handling user registration (POST /register).
* Main Page Route: Route for rendering the main page (GET /) if the user is authenticated.
* Logout Route: Route for logging out the user (GET /logout).

**File Upload**

* Upload Route: The /upload route handles file uploads, supporting both text input and file uploads.
* Multer Middleware: Multer middleware is used to parse multipart/form-data requests and handle file uploads.

**Socket.io Event Handling**

* Model Response Events: Event handlers are set up to handle model response events and emit notifications to clients.

**Exporting the Server**

* Server Export: The configured Express server is exported to enable its use in other modules.

**WebSockets** is a communication protocol that provides full-duplex communication channels over a single TCP connection. It enables real-time, bidirectional communication between a client (such as a web browser) and a server.

Here are some key features of WebSockets:

* Full-Duplex Communication: Unlike traditional HTTP requests, which follow a request-response model, WebSockets allow both the client and the server to send messages to each other simultaneously, enabling real-time data exchange.
* Low Latency: WebSockets offer low-latency communication, making them suitable for applications that require real-time updates or interactive features, such as chat applications, online gaming, and live dashboards.
* Efficient: WebSockets have a lightweight overhead compared to protocols like HTTP, as they establish and maintain a persistent connection between the client and server, eliminating the need for repeated handshakes for each message exchange.
* Bi-Directional: WebSockets support bidirectional communication, meaning that either the client or the server can initiate a message exchange at any time, without waiting for a request from the other party.
* Protocol Upgrade: The WebSocket protocol starts with an HTTP handshake, during which the client and server negotiate the upgrade to the WebSocket protocol. Once the connection is established, communication switches to the WebSocket protocol.
* WebSockets are commonly used in web applications for implementing features such as real-time chat, live notifications, collaborative editing, and multiplayer gaming. They provide a reliable and efficient mechanism for maintaining persistent, low-latency connections between clients and servers, enabling interactive and responsive user experiences on the web.

WebSockets are commonly used in web applications for implementing features such as real-time chat, live notifications, collaborative editing, and multiplayer gaming. They provide a reliable and efficient mechanism for maintaining persistent, low-latency connections between clients and servers, enabling interactive and responsive user experiences on the web.

# Limitations and Future Improvements

## Limitations

* Dependency on cloud services which may cause latency
* Dependency on models which takes time to start and extract content

## Improvements

* Handling of invalid extensions/ Handling of invalid or not supported formats
* Optimization of time response/latency