API Documentation

# Gemini API Documentation

## I. Overview

Gemini API Purpose: The Google Gemini API, accessed through the google-generativeai Python library, provides access to Google's generative AI models, specifically designed for text generation, multimodal interaction, and structured output.

Key Capabilities (Relevant to DACS):

* Generating realistic and diverse text for client scenarios.
* Handling multi-turn text-based conversations.
* Creating structured output (like JSON) based on provided schemas.
* Incorporating multimedia (images, audio, video) into prompts (though mostly for future iterations).
* Providing text embeddings.

Link to Gemini API Documentation: [URL]

Link to google-generativeai library documentation: [URL]

## II. Authentication

Authentication Method: The Gemini API requires authentication via a Google Cloud API key.

Obtaining the API Key:

* Create a Google Cloud Project or select an existing one.
* Enable the Gemini API.
* Create a new API key in the Credentials section.
* Note: If deploying using Vertex AI, you'll be required to use a service account instead, and authenticate using Oauth.

Usage: For local development, set up your API key as an OS environment variable with the name GOOGLE\_API\_KEY. For production, use Google Cloud Secret Manager.

Code Example:

from google import genai

import os

GOOGLE\_API\_KEY = os.getenv('GOOGLE\_API\_KEY')

genai.configure(api\_key=GOOGLE\_API\_KEY)

model = genai.GenerativeModel('gemini-1.5-pro-latest')

## III. Models

Primary Model (For Phase 1): gemini-1.5-pro-latest

Notes: This is the default model for text and multimodal content generation, and will be used for generating the text for your client scenarios and data description for your datasets.

Other Models:

* gemini-1.5-flash-8b
* gemini-1.5-flash-002
* gemini-1.5-pro-002
* text-embedding-004
* are other possible options to explore in the future but not for this iteration.

All other models are experimental and their outputs should not be used for production.

The SDK allows to get a list of all available models with: client.models.list()

## IV. Text Generation Endpoint (client.models.generate\_content)

Purpose: This endpoint is used to generate text responses based on provided prompts.

Method: client.models.generate\_content(model, contents, config=None)

Parameters:

* model (required): Specifies which Gemini model to use, use gemini-1.5-pro-latest.
* contents (required): The prompt for text generation (can be a string, an image, or a combination of both, with the types.Content method).
* config (optional): Configuration options (using types.GenerateContentConfig), including:
* temperature (float): Controls randomness (0.0-1.0). 0 is deterministic, 1 is very creative.
* top\_p (float): Controls the cumulative probability of the most probable tokens.
* top\_k (integer): Chooses a selection of the most probable tokens.
* max\_output\_tokens (integer): Maximum number of tokens in the response (default is 2048).
* stop\_sequences (list of strings): Sequences that stop generation when encountered.
* presence\_penalty (float): Penalizes repeated tokens.
* frequency\_penalty (float): Penalizes frequent tokens.
* safety\_settings (list of types.SafetySetting objects): Filters to block harmful or unwanted content.
* response\_mime\_type (string): Set to application/json to get JSON output.
* response\_schema (string or pydantic object): Set to constraint the model to return a structured output.
* tools: Enables function calling and provides available functions to the model.
* cached\_content: Use a previously created cache using its name.
* system\_instruction (string): Initial instructions to provide context for the responses

Response Format:

* The response contains the generated text from the model using the property .text. The responses are in JSON format and can include:
* The generated text in the response.text or in response.candidates[0].content.parts[0].text
* The full response data, may include usage metadata (response.usage\_metadata), code information, or functions called.

## V. Multimodal Input:

Content Input: The contents parameter can accept multiple parts, you can also include files, by preparing them before making the call.

* You can use the types.Part.from\_uri() method to include files using their uri, and must specify the mime\_type.

Image Input: Load the image as bytes, then use that information to create a types.Part, specifying that the mime\_type is of type image/jpeg or image/png.

Text Input: Use strings for the text parts, or use the types.Part.from\_text() to explicitly define a text part.

PDF Files: Upload them using client.files.upload(path="path\_to\_pdf"), and use the returned uri and the mime\_type for creating the Part.

## VI. Other Important Endpoints:

* client.models.count\_tokens: Calculates the number of input tokens for a given prompt.
* client.files.upload: Uploads files (images, audio, video, PDFs, etc.) to the Gemini API.
* client.files.get: Get information about a specific file (including its state to make sure that it is uploaded correctly and is ready to use).
* client.caches.create: Creates a cache to save content for faster and more efficient access in subsequent calls.
* client.caches.delete: Deletes cached content.
* client.aio.models.generate\_content: The async version of the text generation endpoint, useful for making asynchronous requests.
* client.models.embed\_content: Generates text embeddings (vector representations of text).
* client.tunings.tune: Used to create new fine-tuning models.
* client.tunings.distill: Used to distill models.

## VII. Error Handling (API specific):

Gemini API Errors:

* Use try-except blocks to catch exceptions that can be raised by the API or any processing step.
* Log errors using Python's logging module.
* Handle API errors gracefully to ensure the application doesn't crash. Provide a user-friendly error message when a response cannot be obtained.
* Specific errors can be caught by checking for specific HTTP status codes or by checking the type of the error.