AI API Server

> **Audience:** Internal engineers

Introduction

This service provides five core AI-driven endpoints— `/generate_caption` , `/catalog-ai` , `/create_order` , `/generate-images` , and `/regenerate-image` . Each handles structured inputs (file uploads or JSON), leverages our Gemini/CLIP pipelines, and returns JSON or file bundles. Infrastructure details (auth, S3, Milvus) are abstracted; this README focuses on endpoint contracts.

Table of Contents

API Endpoints

1. `POST /generate_caption`

Purpose: Analyze a jewelry image, perform similarity search, generate metadata (quality, name, description, attributes, color) and a one-line e-commerce caption.

description: The `/generate_caption` endpoint accepts a multipart upload consisting of an image file and a style **type** (e.g., “Elegance”, “Boldness”, “Artistry”). It first queries our image_search_engine to detect whether this exact product image already exists in S3; if a match is found, it returns the existing S3 URL. Otherwise, it saves the image locally, uploads it to S3. then streams the image into Gemini to generate a JSON payload containing a one-line caption, quality grade (A/B/C), a unique name, detected color, and a list of physical attributes. In the case of duplicates, the `"duplicate"` flag is added and the original S3 URL is returned—ensuring we never store redundant images. ```

Request: `multipart/form-data`

- `file` : UploadFile (image/jpeg, image/png, image/webp)
- `type` : string — style category (e.g., `Elegance` , `Artistry`)

Response: `application/json`

```
{
  "display_name": "filename.jpg",
  "generated_name": "Unique Jewelry Name",
  "quality": "A|B|C",
  "description": "One-line caption",
  "attributes": "Physical feature summary",
  "prompt": "Internal prompt used",
  "color": "Detected color",
  "s3_url": "https://...",
  "duplicate": "duplicate found"
}
```

---

2. `POST /catalog-ai`

Purpose: Generate catalog entries and Excel file for jewelry images, formatted per marketplace.

description: Accepts a JSON payload conforming to the `CatalogRequest` schema—containing comma-separated image URLs, SKUs, `type` (ear|nec|bra) and `marketplace` (flipkart|amazon|meesho). The service:

1. Downloads each image and prepares it via our `input_image_setup` utility.
2. Calls Gemini `get_gemini_response` with marketplace- and product-specific `prompts` to extract descriptions, structured attributes, and dimensions.
3. Merges AI-generated data with static `fixed_values` for that format.
4. Streams results into an in-memory Excel template, applies the appropriate writer from `excel_helpers` , and re-uploads the finished `.xlsx` to S3.

Request: `application/json` matching `CatalogRequest`

```
{
  "image_urls": "comma,separated,S3_URLs",
  "type": "ear|nec|bra",
  "marketplace": "flipkart|amazon|meesho",
```

```
  "skuids": "comma,separated,SKUs"
}
```

Response: `application/json`

```
{
  "results": [
    {
      "attributes": {
        "Name": "Golden Dangle Earrings",
        "Color": "Gold",
        "Base Material": "Brass",
        "Stone Type": "Cubic Zirconia",
        "Clasp Type": "Push Back",
        "Plating": "18K Gold"
      }
    },
    ...
  ],
  "excel_file": "https://.../earrings_flipkart.xlsx"
}
```

---

3. `POST /create_order`

Purpose: Parse a PDF invoice, extract structured metadata and product list using Gemini.

Request: `multipart/form-data`

- `file` : UploadFile (application/pdf)

Response: `application/json`

```
{
  "parsed_data": {
    "order_id": "ORD1234",
    "vendor": "ABC Jewels",
    "total_amount": "₹5000",
    "invoice_date": "2025-06-15"
  },
  "products_data": [
    {
      "product_price": "1500",
      "item_type": "Ring",
      "sku_id": "RGX999",
      "quantity": "2"
    },
    ...
  ]
}
```

---

4. `POST /generate-images`

Purpose: Generate stylized variants of a jewelry image, store in S3, return ZIP URL and individual links.

description: Accepts a JSON body matching the [`ImageRequest`] schema—with an S3 URLs in csv format(`s3_urls`) and a `product_type` (ear|bra|nec). The endpoint:

1. Downloads the original image from the specified S3 URL.
2. Applies a set of product-specific prompts to generate styled variants via our GenAI pipeline.
3. Resizes and uploads each generated image back to S3.
4. Uses `create_zip_from_s3_urls` to bundle all generated image URLs into a ZIP file and re-upload it.

Request: `application/json` matching `ImageRequest`

```
{
  "s3_urls": "https://s3.amazonaws.com/bucket/img123.jpg",
  "product_type": "ear"
}
```

Response: `application/json`

```
{
  "zip_url": "https://s3.amazonaws.com/bucket/img123.zip",
  "original_image_url": "https://s3.amazonaws.com/bucket/img123.jpg",
  "gen_images": [
    { "prompt_index": 0, "image_url":
"https://s3.amazonaws.com/bucket/img123_v1.jpg" },
    { "prompt_index": 1, "image_url":
"https://s3.amazonaws.com/bucket/img123_v2.jpg" },
    ...
  ]
}
```

---

5. `POST /regenerate-image`

Purpose: Regenerate a specific variant image and patch existing ZIP + S3 path with updated version.

description: This endpoint updates a single generated image and synchronously patches it within the associated ZIP archive on S3.

It accepts a multipart form containing the original image URL, the S3 path of the image to be replaced, the current ZIP URL, the product type, and the prompt index to re-apply. The endpoint:

1. Validates the `product_type` and `prompt_index` against supported GenAI prompt maps.
2. Downloads the original image, regenerates a single variant using the specified prompt.
3. Replaces the old image in S3 and updates the corresponding ZIP using the helper function `update_zip_on_s3`, preserving all other ZIP contents.
4. Returns the same image URL (now with updated contents) and the original ZIP URL, now containing the new file.

Request: `multipart/form-data`

- `original_image_url` : string
- `old_generated_url` : string
- `old_zip_url` : string
- `product_type` : string
- `prompt_index` : integer

Response: `application/json`

```json
{
  "image_url": "https://s3.amazonaws.com/bucket/img123_v1.jpg",
  "zip_url": "https://s3.amazonaws.com/bucket/img123.zip"
}
```

---

utils.py

utility functions for catalog ai endpoint

1. `get_gemini_responses`

Invokes the Gemini model (`gemini-2.0-flash`) to process a given image and generate textual responses for multiple prompts.

- **Signature**

```
get_gemini_responses(input: str, image: Tuple[bytes, str], prompts:
List[str]) -> List[str]
```

- **Parameters**
  - `input` (`str`): A base string instruction or context shared across all prompts.
  - `image` (`Tuple[bytes, str]`): A tuple containing the image file (typically `image[0]` is passed to the model).
  - `prompts` (`List[str]`): A list of prompt strings to be used individually for generating content.
- **Workflow**
  1. **Model Initialization**
     A `GenerativeModel` is created using `gemini-2.0-flash` with a specific `generation_config` :
     - `temperature=0.5` : Controls randomness; lower values → more deterministic.
     - `top_p=0.9` & `top_k=40` : Sampling techniques to limit output to top probable tokens.
     - `max_output_tokens=1024` : Limits the length of generated output.
  2. **Prompt Processing Loop**
     Iterates over each prompt in `prompts` :
     - Calls `generate_content()` on the model with `[input, image[0], prompt]` as the multi-modal input.
     - Collects the `text` parts of the response into a list.
  3. **Response Collection**
     - Each response may contain multiple `parts` (especially if the model returns segmented content).
     - Only `part.text` (non-empty textual parts) are appended to `all_responses` .
- **Returns**
  - A list of strings where each string is a generated response corresponding to each prompt.
- **Usage Tip**
  - Ideal for multi-prompt image interpretation tasks (e.g., describing an image under different styles or answering questions based on visual input).

---

2. `get_gemini_dims_responses`

similar to get_gemini_responses only difference is a (thinking/more advanced) model is used to get dimensions with higher max output tokens. this should not be used for other task as it has lower free tier limit and a higher cost.

3. `input_image_setup`

- **Signature**: `input_image_setup(file_bytes_io: io.BytesIO, mime_type: str) -> List[Dict[str, bytes]]`
- Raises `FileNotFoundError` if `file_bytes_io` is `None`.
- Extracts raw bytes with `file_bytes_io.getvalue()`.
- Wraps bytes in `[{"mime_type": mime_type, "data": bytes_data}]`.
- Returns `image_parts` ready for the Gemini API.

4. `write_to_excel_amz_xl`

Writes structured product data into an Excel file, creating headers if needed or appending to an existing sheet.

- **Signature**

  ```
  write_to_excel_amz_xl(
      results: List[Tuple[str, Dict[str, Any], str]],
      filename: str,
      target_fields: List[str],
      fixed_values: Dict[str, Any]
  ) -> None
  ```

- **Parameters**
  - `results`:
    A list of tuples `(image_name, response, description)` where
    - `image_name` (`str`) is the filename of the image.
    - `response` (`Dict[str, Any]`) maps each target field to its generated value.
    - `description` (`str`) is a free-text product description.
  - `filename` (`str`): Path to the Excel file to load or create.
  - `target_fields` (`List[str]`): Fields to populate from `response`.
  - `fixed_values` (`Dict[str, Any]`): Constant field-value pairs to inject on every row.
- **Workflow**
  1. **Load or Initialize Workbook**
     - If `filename` exists, load it and grab the first worksheet.
     - Otherwise, create a new `Workbook`, set up a default header row, and build a header-to-column map.
  2. **Determine Columns**
     - Map header names → column indices from the header row.
     - Combine `target_fields` with keys of `fixed_values` to form `all_fields`, then filter by headers present.
  3. **Find Next Empty Row**

        - Scan down column 1 until an empty cell; that row index is the insertion point.
    4. **Write Data Rows**
        - For each `(image_name, response, description)`:
            - Merge `response` with `fixed_values`.
            - Add `"product_description": description` and `"item_sku": os.path.splitext(image_name)[0]`.
            - Write each field value into its mapped column on the current row.
            - Increment row index.
    5. **Save Workbook**
        - Call `wb.save(filename)`.
- **Returns**
    - `None` (writes directly to `filename`).

5 `write_to_excel_flipkart` and `write_to_excel_meesho`

uses same logic as above function only some parameter are different like sheet number , column number to check for empty row , target field names and fixed fields dict.

excel_fields.py

contains target fields list and fixed values dict for each product type and market place
target fields -> fields which are generated by gemini

1. to add a fixed value just append the key value in its respective dict , the key name should match the column name in the excel file
2. to add a target value add the column name of excel file in the list also change in the respective prompt as defined in `prompts.py

prompts.py

contains the prompts for catalog ai endpoint each prodcut marketplace combination has three prompts

1. questions prompt
2. description prompt: for description field
3. dimensions prompt
   if new target field is added then a new question should be added in the questions prompt in the following format

```
  {
      "field name": "should match the column name as in excel sheet and as in target column list",
      "question": "explain what ai needs to identify and give output of",
      "options": "dropdown options if any (leave empty if none)"
  }
```

utils2.py

contains utility functions for image generation api endpoint

this uses 4 api keys of google as one api key provide only 100 images per day the generate_with_failover function calls gemini with one api key, it uses is_quoata_excedded function to check if the api key is giving a valid response if a quota excedded error response comes it swithces the key and checks it again

1. `get_gemini_responses`

Invokes the Gemini model (`gemini-2.0-flash-preview-image-generation`) to process a given image and generate textual responses for multiple prompts.

- **Signature**

  ```
  get_gemini_responses(input: str, image: Tuple[bytes, str], prompts:
  List[str]) -> List[str]
  ```

- **Parameters**
  - `input` (`str`): A base string instruction or context shared across all prompts.
  - `image` (`Tuple[bytes, str]`): A tuple containing the image file (typically `image[0]` is passed to the model).
  - `prompts` (`List[str]`): A list of prompt strings to be used individually for generating content.
- **Workflow**
  1. **Model Initialization**
     A `GenerativeModel` is created using `gemini-2.0-flash-preview-image-generation` with a specific `generation_config` of 'TEXT' and 'IMAGE':
  2. **Prompt Processing Loop**
     Iterates over each prompt in `prompts` :
     - Calls generate_with_failover for each prompt
  3. **Response Collection**
     - prepares structured response
     - collects image bytes generated which then can be rendered and used
- **Returns**
  - Returns a List[Dict] where each dict has: "prompt": the prompt used, "text": combined text output , "images": list of image bytes

2. `resize_image and resize_image_2`

resiszes the images to 2000X2000 pixels and 1080X1440 pixels respectively

prompts2.py

contains the prompts for image generation the first 5 prompts are for earrings image generation. the next 5 are for bracelet and the last 6 are for necklace.
for earrings and braclet 5 images are generated and for necklace 6 images are generated
to change the prompt only this file needs to be edited

image_search_engine.py

uses milvus db to store embeddings and perform similarity search
zilliz cloud (zilliz.com) is used to store embeddings , it takes URL and TOKEN from the enviroment variable to connect to the cloud db.
uses images from s3 bucket which needs:

1. S3_BUCKET - bucket name
2. aws_access_key_id - set as environment variable
3. aws_secret_access_key - set as environment variable

1. `init_clip`

this function initializes the open ai open source image embeddings model CLIP which is used to create vector embeddings from images
model used - openai/clip-vit-base-patch32

2. `init_milvus`

this initializes the milvus cluster if it is not already present in the cloud. the cluster schema will habe two fields one is unique id and other is the vector embedding of the image.

3. `get_image_paths_from_s3`

this calls the get paginator function of s3 to get all the paths (URLS) of images stored it removes the paths with /gen_images (stores ai gen images) and
/excel_files (stores excel files used in catalog ai).

returns a list of all image paths (only paths are returned not the actual images)

4. `index_images_from_s3`

Indexes embeddings for images stored in S3 into a Milvus collection, and maintains hash-to-path mappings.

Parameters

- `collection` : Milvus collection object where embeddings will be inserted/deleted.

- `image_keys` (`List[str]`): List of S3 object keys (paths) for images to index.
- `model` : CLIP or similar model instance used for embedding.
- `processor` : Preprocessing object (e.g. CLIP processor) for preparing images.
- `device` : PyTorch device ("cpu" or "cuda") for running inference.
- `batch_size` (`int` , default 128): Number of embeddings to accumulate before bulk-inserting.
- `index_file` (`str`): Path to JSON file storing previously indexed image hashes.
- `int_hash_file` (`str`): Path to JSON file mapping integer hashes → public S3 URLs.

Workflow

1. **Load existing index**
   - If `index_file` exists, read its JSON into `indexed` (mapping of image hash → key).
2. **Initialize batches & mappings**
   - `batch_ids` , `batch_embeddings` collect IDs and vectors for bulk insert.
   - `current_hashes` will map each image's hash → key.
   - `int_hash_to_path` will map integer hash → public S3 URL.
3. **Iterate over `image_keys`**
   For each `key` in `image_keys` :
   1. Compute `h = hash_image_from_s3(...)` and convert to integer `int_h` .
   2. Record `current_hashes[h] = key` and `int_hash_to_path[str(int_h)] = public_url` .
   3. **Skip** if `h` already in `indexed` . (does not reindex images with vecotrs already present)
   4. Otherwise:
      - Download & load image (`load_image_from_s3`).
      - Compute embedding (`embed_image`).
      - Append `int_h` to `batch_ids` , and vector to `batch_embeddings` .
   5. Once `batch_ids` reaches `batch_size` , bulk-insert into `collection` :

      ```
      entities = [{"id": i, "embedding": e} for i,e in zip(batch_ids,
      batch_embeddings)]
      collection.insert(entities)
      ```

      - Reset batch lists.
4. **Flush remaining batch**
   - After loop, if any `batch_ids` remain, insert them similarly.
5. **Delete stale embeddings**
   these are deleted images from s3 which have been indexed before but now not present

in s3, so there vectors are deleted from vector db
   - Compute `deleted_hashes = set(indexed.keys()) - set(current_hashes.keys())`.
   - Convert to integer IDs and issue `collection.delete(expr=...)`.
6. **Flush & reload collection**
   - If any new inserts/deletions occurred, call `collection.flush()` and `collection.load()` to apply changes.
7. **Persist mapping files**
   - Overwrite `index_file` with `current_hashes`.
   - Overwrite `int_hash_file` with `int_hash_to_path`.
8. **Logging**
   - Prints status messages at each major step, including counts and success/failure markers.

---

5. embed_image

used to generate embedings of query image for which search needs to be performed

6. search_similar

calls the search function if milvus db to get the search results and the distance values of top 5 mathces.

deployment

1. first push the code to your github
2. the api server is deployed on ec2 , to login into ec2 terminal you will need the permission file (.pem) without which it can not be accesed and the public ip address of the ec2. use ssh command to login into the ec2 vm from your machine
3. once inside the ec2 , go to the project directory , set git remote origin to your git hub then pull code from your github using `git pull origin main`
4. terminate the running uvicorn server
5. export the environment variables
6. using nohup start the server again
7. the .json hashed files should not be commited or pulled from github as it will delete the original hashes and will cause problems in image search.
8. google can change its model names or depreciate the model accordingly model name should be changed in utils1.py and utils2.py
9. api keys should not be exposed in github and should be directly exported in ec2