ChatGPT

Google Gen AI Python SDK

The **Google Gen AI Python SDK** provides a unified interface for Google's generative AI services (Gemini/Vertex AI). It supports both the Gemini Developer API and the Gemini API on Vertex AI, allowing developers to prototype on one platform and migrate to the other without major code changes [1]. The SDK is available on PyPI (`google-genai`, Python ≥3.9) and GitHub. Below is a comprehensive guide to using the library, including installation, configuration, and every API surface.

Installation & Setup

- **Install** via pip:

```
pip install --upgrade google-genai
```

(You may also use alternative pip executables, e.g. `uv pip install google-genai` [2].)

- **Imports:** In your code, import the SDK modules as:

```
from google import genai
from google.genai import types
```

This provides access to the `genai` client and the `types` namespace for request/response data models [3].

- **Environment Variables:** The client can also read credentials and configuration from environment variables:

- For the **Gemini Developer API**, set either `GEMINI_API_KEY` or (preferably) `GOOGLE_API_KEY`:

```
export GEMINI_API_KEY="your-gemini-key"
```

- For the **Gemini API on Vertex AI**, set `GOOGLE_GENAI_USE_VERTEXAI=true` plus your Google Cloud project/location:

```
export GOOGLE_GENAI_USE_VERTEXAI=true
export GOOGLE_CLOUD_PROJECT="your-project-id"
export GOOGLE_CLOUD_LOCATION="us-central1"
```

The client will automatically pick up these variables. If both `GEMINI_API_KEY` and `GOOGLE_API_KEY` are set, `GOOGLE_API_KEY` takes precedence [4] [5].

Client Initialization

Create a client object as your entry point to the SDK:

- **Gemini API (server-side)**: provide `api_key`:

```
client = genai.Client(api_key="GEMINI_API_KEY")
```

- **Vertex AI**: use `vertexai=True` with a project and location (and credentials via ADC or env vars):

```
client = genai.Client(vertexai=True, project="your-project-id",
location="us-central1")
```

- **Express (client-side) mode on Vertex AI**: a hybrid mode using an API key:

```
client = genai.Client(vertexai=True, api_key="YOUR_API_KEY")
```

All initialization options support an optional `http_options` parameter (an instance of `types.HttpOptions`) to configure HTTP settings such as API version, timeouts, and custom body fields. For example, to use the stable API version ("v1" rather than the default beta), pass:

```
client = genai.Client(
    vertexai=True, project="your-project-id", location="us-central1",
    http_options=types.HttpOptions(api_version="v1")
)
```

For the Gemini API, you can set `api_version='v1alpha'` in `HttpOptions` [6] [7].

HTTP and Transport Options

- **Sync vs Async Clients:** The `Client` object is synchronous by default. To use async methods, access `client.aio`, which exposes all the same services as async coroutines. E.g. `await client.aio.models.generate_content(...)` [8].

- **Closing Clients:** Always close the client to free resources. For the sync client, call `client.close()` after use [9]. For the async client, use `await client.aio.aclose()` [10].

- **Context Managers:** Both sync and async clients support context managers:

```
from google.genai import Client

with Client() as client:
    # use client normally; it will be closed on exit
    result = client.models.generate_content(...)
```

```
async with Client().aio as aclient:
    # use async client in an async loop
    resp = await aclient.models.generate_content(...)
```

The context manager automatically closes the underlying client upon exit [11] [12] .

- **HTTP Backend (aiohttp):** By default the SDK uses `httpx` under the hood. For better performance in async mode, you can install the `aiohttp` extra (`pip install google-genai[aiohttp]`). You can pass aiohttp-specific args via `HttpOptions(async_client_args={...})` [13] . E.g.:

```
from google.genai import types
http_options = types.HttpOptions(async_client_args={'cookies': ...,
'ssl': ...})
client = genai.Client(..., http_options=http_options)
```

(By default `trust_env=True` is set to respect env-var proxies as `httpx` does [13] .)

- **Proxy Settings:** Both `httpx` and `aiohttp` respect standard environment proxy vars. You can set `HTTPS_PROXY` and `SSL_CERT_FILE` before client init:

```
export HTTPS_PROXY="http://user:pass@proxy:port"
export SSL_CERT_FILE="client.pem"
```

For SOCKS5, use `httpx[socks]` and pass the `proxy` in `HttpOptions.client_args`:

```
opts = types.HttpOptions(
    client_args={'proxy': 'socks5://user:pass@host:port'},
    async_client_args={'proxy': 'socks5://user:pass@host:port'}
)
client = genai.Client(..., http_options=opts)
```

[14] [15] .

- **Custom Base URL:** You can override the service base URL (e.g. to use an API gateway). Pass it in `HttpOptions`:

```
client = genai.Client(
    vertexai=True,
    http_options={
        'base_url': 'https://custom-api-gateway.com',
        'headers': {'Authorization': 'Bearer token'}
    }
)
```

(This also allows bypassing project/location checks on the client side) [16].

Types and Data Models

The SDK uses Python **TypedDicts** and **Pydantic** models (in `google.genai.types`) to represent API messages. You can supply input parameters either as plain dicts or as instances of these Pydantic classes. For example, many methods accept a `config` parameter where you can pass either a dict or a `types.GenerateContentConfig` object [17] [18].

Important types include:

- **Content & Part:** The `contents` argument for text/image generation methods must be a list of `Content`-like items. Each `Content` consists of a `role` (`"user"` or `"model"`) and a list of `Part` instances. You can build parts from text, files, URIs, function calls, etc. Helper methods include `types.Part.from_text()`, `from_bytes()`, `from_uri()`, `from_function_call()`, `from_function_response()`, etc. The SDK will auto-convert simpler inputs into `Content` instances (see below).

- **Function/Tool Declarations:** For function calling, `types.FunctionDeclaration`, `types.Tool`, and `types.AutomaticFunctionCallingConfig` let you declare and invoke functions. (E.g. to call a Python function automatically, include it in the `tools` list as shown below [19] [20].)

- **Configuration Objects:** Common config classes include:

- `GenerateContentConfig` (params like `temperature`, `max_output_tokens`, `stop_sequences`, etc).
- `EmbedContentConfig` (for embedding output dimensionality, etc).
- `GenerateImagesConfig`, `UpscaleImageConfig`, `EditImageConfig` (for image model parameters like `num_images`, `output_mime_type`, etc).
- `GenerateVideosConfig` (for video generation).
- `CreateCachedContentConfig` (for caching settings).
- `CreateTuningJobConfig`, `UpdateModelConfig`, etc (for fine-tuning jobs).
- `ListBatchJobsConfig`, `UploadFileConfig`, etc.

You can find all config and message classes in `google.genai.types`. Example:

```
from google.genai import types
config = types.GenerateContentConfig(temperature=0.3, max_output_tokens=100)
```

and pass it to `client.models.generate_content(model="gemini-2.5-flash", contents="Hi", config=config)` [21] [18].

Providing `contents` for Text/Image Generation

The `client.models.generate_content()` method (and related functions) accepts a `contents` parameter that can be specified in many ways. Internally the SDK converts any valid input into a **list of** `types.Content`. Key input forms:

- **Single string:** e.g. `contents="Hello"`. This is treated as one `types.UserContent` with one text part (role=`user`). Equivalent to `[types.UserContent(parts=[types.Part.from_text("Hello")])]` [22].
- **List of strings:** e.g. `contents=["Q1", "Q2"]`. The SDK groups multiple strings into *one* `UserContent` (all parts in one content). In effect:

  ```
  [
      types.UserContent(parts=[
          types.Part.from_text("Q1"),
          types.Part.from_text("Q2")
      ])
  ]
  ```

  (Role is `"user"`) [23].
- **Pydantic** `types.Content` **instance:** e.g.

  ```
  contents = types.Content(
      role="user",
      parts=[types.Part.from_text("Why is the sky blue?")]
  )
  ```

  The SDK will wrap it in a list if needed (resulting list of one content) [24].
- **List of** `types.Content`**:** Pass a Python list of `Content` objects directly – the SDK will use it as-is.

- **Function call part(s):** Use `types.Part.from_function_call(name, args)`. A single `Part` produces a `ModelContent` by default (role=`model`). Multiple function-call parts yield one `ModelContent` containing them [25]. E.g.:

```
parts = [types.Part.from_function_call(name="get_weather",
args={"location":"Boston"})]
contents = [types.ModelContent(parts=parts)]
```

  or just give the `Part` directly (SDK will create the wrapping content).

- **URI/Bytes parts (for images/audio):** Use `types.Part.from_uri(file_uri, mime_type)` or `from_bytes(data, mime_type)`. A single non-function part yields a `UserContent`, multiple parts of mixed types are grouped by the SDK (see docs) [26] [27].

- **Mixed lists:** You can mix `Content` objects and raw `Part` objects. The SDK will combine consecutive non-function-call parts into one `UserContent` and consecutive function-call parts into one `ModelContent` [28].

See the detailed examples below. In all cases, the SDK normalizes your input to `list[types.Content]` before sending the request [29] [28].

Models: Inference & Management

The `client.models` service provides **model inferencing** and **model management** methods:

Generate Content (Text/Image/JSON)

- **Text Generation (synchronous):**

```
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Tell me a joke."
)
print(response.text)
```

  This sends a text prompt to the model and returns a `response` with fields such as `.text` and `.candidates` [30].

- **Text-to-Image (synchronous):** Use a text prompt with a vision model:

```
from google.genai import types

response = client.models.generate_content(
```

```
    model="gemini-2.5-flash-image",
    contents="A futuristic cityscape at sunset",
    config=types.GenerateContentConfig(
        response_modalities=["IMAGE"],
        image_config=types.ImageConfig(aspect_ratio="16:9")
    ),
)
for part in response.parts:
    if part.inline_data:
        img = part.as_image()
        img.show()
```

Here we request an image output (`IMAGE` modality) and specify image dimensions in `ImageConfig` [31].

- **File-based Generation (Gemini API only):** Upload a file and have the model process it:

```
!wget -q https://storage.googleapis.com/generativeai-downloads/data/a11.txt
```

```
file = client.files.upload(file="a11.txt")
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents=["Summarize this file.", file]
)
print(response.text)
```

The `contents` list mixes a prompt string with a `File` object. The SDK reads the file and includes it in the request [32] [33].

- **Configuring Generation:** Use `types.GenerateContentConfig` to tune parameters. For example, to set a system instruction, max tokens, and temperature:

```
from google.genai import types

response = client.models.generate_content(
    model="gemini-2.0-flash-001",
    contents="Hello",
    config=types.GenerateContentConfig(
        system_instruction="You are a helpful assistant.",
        max_output_tokens=50,
        temperature=0.0
    )
```

```
)
print(response.text)
```

Lowering `temperature` makes outputs more deterministic. All available parameters (like `top_p`, `stop_sequences`, etc.) can be found in the Vertex AI/Gemini docs [34] [21].

- **Pydantic vs Dict Config:** You can pass the `config` as a dict or as a Pydantic model. Both work. Example using typed config:

```
response = client.models.generate_content(
    model="gemini-2.0-flash-001",
    contents="Why is the sky blue?",
    config=types.GenerateContentConfig(
        temperature=0,
        top_p=0.95,
        max_output_tokens=100,
        stop_sequences=["STOP!"]
    )
)
```

This is equivalent to providing a dict `config={"temperature": 0, "top_p": 0.95, ...}` [18].

- **Batch Model Listing:** Retrieve available base models:

```
for model in client.models.list():
    print(model.name)
```

You can use paging via `page_size` and iterators. An async iterator is also available (`await client.aio.models.list()` yields models) [35] [36].

- **Safety Settings:** Add `safety_settings` in the config to block undesired content. Example blocking hate speech:

```
from google.genai import types
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Say something offensive.",
    config=types.GenerateContentConfig(
        safety_settings=[
            types.SafetySetting(
                category="HARM_CATEGORY_HATE_SPEECH",
                threshold="BLOCK_ONLY_HIGH"
            )
```

```
        ]
    )
)
print(response.text)
```

This example sets a threshold to block high-level hate speech [37].

- **Function Calling (Auto-invoke Python functions):** By default, you can pass a Python function directly in `tools` and the SDK will invoke it if the model emits a function call. Example:

```
from google.genai import types

def get_weather(location: str) -> str:
    return "sunny"

response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="What's the weather in Boston?",
    config=types.GenerateContentConfig(tools=[get_weather])
)
print(response.text)  # model query -> function call -> prints function
result
```

The function is declared automatically, and the model's function call response is executed [19].

- **Disabling Automatic Function Calling:** If you want to prevent the SDK from auto-calling your Python function (and just return the function call parts for you to handle manually), use `automatic_function_calling=types.AutomaticFunctionCallingConfig(disable=True)`:

```
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="What's the weather in Boston?",
    config=types.GenerateContentConfig(
        tools=[get_weather],

automatic_function_calling=types.AutomaticFunctionCallingConfig(disable=True)
    )
)
# You can then inspect response.function_calls to get the model's call.
print(response.function_calls)
```

This will populate `response.function_calls` instead of invoking `get_weather` [38].

- **Manual Function Declaration:** For more control, declare a function schema manually using `FunctionDeclaration` and wrap in a `Tool`:

```
from google.genai import types

func_schema = types.FunctionDeclaration(
    name="get_weather",
    description="Get current weather",
    parameters_json_schema={
        "type": "object",
        "properties": {"location": {"type": "string"}},
        "required": ["location"]
    }
)
tool = types.Tool(function_declarations=[func_schema])
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="What's the weather in Boston?",
    config=types.GenerateContentConfig(tools=[tool])
)
print(response.function_calls[0])
```

You then receive a function call part in `response.function_calls`. You can execute the real function, wrap its output in `Part.from_function_response()`, and feed it back in a follow-up call [39] [40].

- **Function Calling (Any Mode):** If you set function calling mode to `"ANY"`, the model will **always** generate function calls. You can limit the number of auto-calls via `maximum_remote_calls`. Example (allow 1 auto-call):

```
from google.genai import types

# Disable immediate call after ANY mode
config = types.GenerateContentConfig(
    tools=[get_weather],

automatic_function_calling=types.AutomaticFunctionCallingConfig(maximum_remote_calls=2),
    tool_config=types.ToolConfig(
        function_calling_config=types.FunctionCallingConfig(mode="ANY")
    )
)
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Weather in Boston?",
```

```
    config=config
)
```

Here the model will attempt up to 1 function call (ANY mode + max calls=2) [41] [42].

- **Model Context Protocol (MCP) [Experimental]:** You can integrate a local MCP server as a `tool`. For example, using `mcp-client` (not part of this SDK), you can stream audio/video to a Gemini Live session. An example setup with asyncio and websockets is shown in the GenAI docs [43] [44]. This is an advanced use case (see Gemini Live API docs).

- **JSON/Pydantic Response Schemas:** You can enforce structured JSON output from the model by providing a JSON schema or Pydantic model in the config:

- *JSON Schema:*

```
user_profile_schema = {
    "type": "object",
    "properties": {
        "age": {"type": "integer"},
        "username": {"type": "string"}
    },
    "required": ["age","username"]
}
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Create a random user profile.",
    config={"response_mime_type":"application/json",
"response_json_schema": user_profile_schema}
)
print(response.parsed)  # returns a parsed dict per schema
```

See [51] for an example in docs [45] [46].

- *Pydantic Schema:*

```
from pydantic import BaseModel
class CountryInfo(BaseModel):
    name: str; capital: str; population: int; ...
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Give info for USA.",
    config=types.GenerateContentConfig(
        response_mime_type="application/json",
        response_schema=CountryInfo
    )
```

```
)
print(response.parsed)  # returns an instance or dict of CountryInfo
```

(Alternatively pass a JSON schema as a dict to `response_schema` [47] [48].)

- **Enum Response Schema:** You can ask the model to return one of a Python `Enum`. For example:

```
from enum import Enum
class InstrumentEnum(Enum):
    PERCUSSION = "Percussion"; STRING = "String"; ...
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="What instrument plays multiple notes at once?",
    config={"response_mime_type": "text/x.enum", "response_schema":
InstrumentEnum}
)
print(response.text)  # e.g. "Keyboard"
```

Using `"text/x.enum"` returns the enum value in text form [49] [50].

- **Streaming Generation (sync):** Use `generate_content_stream` to iterate over response chunks as they arrive:

```
for chunk in client.models.generate_content_stream(
    model="gemini-2.5-flash",
    contents="Tell me a story in 50 words."
):
    print(chunk.text, end='')
```

The generator yields partial `Response` objects (with `.text` so far) [51].

- **Streaming Image (sync):** If your prompt contains images:

```
from google.genai import types
for chunk in client.models.generate_content_stream(
    model="gemini-2.5-flash",
    contents=[
        "Describe this image:",
        types.Part.from_uri(file_uri="gs://bucket/img.jpg",
mime_type="image/jpeg")
    ]
):
    print(chunk.text, end='')
```

You can also use `Part.from_bytes()` for local files [52] [53].

- **Asynchronous Generation:** All generation methods have `async` counterparts under `client.aio.models`. Examples:

```
resp = await client.aio.models.generate_content(model="gemini-2.5-flash",
contents="Hi")
async for chunk in await
client.aio.models.generate_content_stream(model="gemini-2.5-flash",
contents="Hello"):
    print(chunk.text)
```

(The `aio` methods are fully analogous to the sync ones [8] [54].)

- **Token Counting:**

- **Count Tokens (server-side):**

```
result = client.models.count_tokens(model="gemini-2.5-flash",
contents="Hello world")
print(result)
```

(This calls the backend to count tokens.)

- **Compute Tokens (local, Vertex only):**

```
result = client.models.compute_tokens(model="gemini-2.5-flash",
contents="Hello world")
print(result)
```

- **Async Count Tokens:** `await client.aio.models.count_tokens(...)`.
- **Local Tokenizer:** Use `genai.LocalTokenizer` to count/compute tokens offline for supported models:

```
tokenizer = genai.LocalTokenizer(model_name="gemini-2.5-flash")
n = tokenizer.count_tokens("Hello")
m = tokenizer.compute_tokens("Hello")
```

These do not require network calls [55] [56].

Image Models (Imagen)

- **Generate Image:** (Gemini API only, allowlisted)

```
from google.genai import types
resp1 = client.models.generate_images(
    model="imagen-3.0-generate-002",
    prompt="A rainy night in a neon city",
    config=types.GenerateImagesConfig(
        number_of_images=1,
        include_rai_reason=True,
        output_mime_type="image/jpeg"
    )
)
img = resp1.generated_images[0].image
img.show()
```

This returns generated images in `resp.generated_images` [57] .

- **Upscale Image:** (Vertex AI only)

```
# Given resp1 from above:
resp2 = client.models.upscale_image(
    model="imagen-3.0-generate-002",
    image=resp1.generated_images[0].image,
    upscale_factor="x2",
    config=types.UpscaleImageConfig(
        include_rai_reason=True,
        output_mime_type="image/jpeg"
    )
)
up_img = resp2.generated_images[0].image
up_img.show()
```

This uses the `imagen` service to upscale an image [58] .

- **Edit Image:** (Vertex AI only) Use a separate model (`*.capability-001`) with reference images:

```
from google.genai.types import RawReferenceImage, MaskReferenceImage

raw_ref = RawReferenceImage(
    reference_id=1,
    reference_image=resp1.generated_images[0].image
)
mask_ref = MaskReferenceImage(
    reference_id=2,
    config=types.MaskReferenceConfig(mask_mode="MASK_MODE_BACKGROUND",
mask_dilation=0)
```

```
)
resp3 = client.models.edit_image(
    model="imagen-3.0-capability-001",
    prompt="Sunlight and clear sky",
    reference_images=[raw_ref, mask_ref],
    config=types.EditImageConfig(
        edit_mode="EDIT_MODE_INPAINT_INSERTION",
        number_of_images=1,
        include_rai_reason=True,
        output_mime_type="image/jpeg"
    )
)
resp3.generated_images[0].image.show()
```

(Here we used a raw image and a mask image as tools to guide the editing [59] [60].)

Video Models (Veo)

- **Generate Video (text → video):** (public preview)

```
from google.genai import types
operation = client.models.generate_videos(
    model="veo-2.0-generate-001",
    prompt="A neon hologram of a cat driving at top speed",
    config=types.GenerateVideosConfig(number_of_videos=1,
duration_seconds=5, enhance_prompt=True)
)
# Poll until done
import time
while not operation.done:
    time.sleep(10)
    operation = client.operations.get(operation)
video = operation.response.generated_videos[0].video
video.show()
```

This is a long-running operation (returns an Operation) [61] [62].

- **Generate Video (image → video):**

```
from google.genai import types
image = types.Image.from_file("path/to/image.png")  # infers MIME type
operation = client.models.generate_videos(
    model="veo-2.0-generate-001",
    prompt="Night sky",
    image=image,
```

```
    config=types.GenerateVideosConfig(number_of_videos=1,
duration_seconds=5, enhance_prompt=True)
)
# Poll as above until done...
video = operation.response.generated_videos[0].video
video.show()
```

(Pass an image for the model to animate [63] [64].)

- **Generate Video (video → video):** (Vertex only)

```
from google.genai import types
video_input = types.Video.from_file("path/to/video.mp4")  # infers MIME
operation = client.models.generate_videos(
    model="veo-2.0-generate-001",
    prompt="Night sky",
    video=video_input,
    config=types.GenerateVideosConfig(number_of_videos=1,
duration_seconds=5, enhance_prompt=True)
)
# Poll...
video = operation.response.generated_videos[0].video
video.show()
```

(Works on an input video URI; requires Vertex [65] [66].)

Model Management

- **Get Base Model:** Retrieve model metadata by name:

```
model = client.models.get(model="gemini-2.5-flash")
print(model.name, model.display_name)
```

- **Update Base Model:** (Rename or describe a base model endpoint)

```
updated = client.models.update(
    model=model.name,
    config=types.UpdateModelConfig(display_name="New Name",
description="Desc")
)
print(updated.display_name)
```

(Useful to label or update model endpoints in Vertex AI) [67].

- **List Tuned Models:** (See Tunings below)

Chats (`client.chats`)

The `client.chats` service manages multi-turn conversations with chat models:

- **Create a Chat Session:**

```
chat = client.chats.create(model="gemini-2.5-flash")
```

This returns a `Chat` (sync) or `AsyncChat` (with `client.aio.chats.create`) object.

- **Send Message (sync):**

```
response = chat.send_message("Tell me a story.")
print(response.text)
```

You can send multiple messages to the same `chat` object to maintain context.

- **Send Message (sync streaming):** Iterate for tokens as they arrive:

```
for chunk in chat.send_message_stream("Tell me a story"):
    print(chunk.text, end="")
```

(This works similarly to `generate_content_stream` but maintains chat history) [68] [69] .

- **Send Message (async):**

```
achat = await client.aio.chats.create(model="gemini-2.5-flash")
response = await achat.send_message("Tell me a story")
print(response.text)
```

- **Send Message (async streaming):**

```
async for chunk in await client.aio.chats.create(model="gemini-2.5-
flash").send_message_stream("Tell me a story"):
    print(chunk.text)
```

(Use `async for` on `send_message_stream` after awaiting the coroutine) [70] .

Files (`client.files`)

(Gemini Developer API only) The `client.files` service lets you manage files used as inputs:

- **Upload File:**

```
file1 = client.files.upload(file="doc1.pdf")
file2 = client.files.upload(file="doc2.pdf")
print(file1.name, file2.name)
```

This returns `File` objects with fields like `.name` (resource name) and `.uri` (gs:// URI) [71].

- **Get File:** Retrieve metadata of an uploaded file:

```
file_info = client.files.get(name=file1.name)
print(file_info)
```

(Requires the file name returned from upload) [72].

- **Delete File:** Delete by name:

```
client.files.delete(name=file1.name)
```

(Also returns a response object) [73].

- **List Files:**

```
for file in client.files.list():
    print(file.name)
```

(Iterator with paging similar to other lists).

Caches (`client.caches`)

The `client.caches` service manages *cached content* (context caching on Vertex AI):

- **Create Cached Content:**

```
from google.genai import types
```

```
uris = [
    "gs://bucket/doc1.pdf",
    "gs://bucket/doc2.pdf"
]
cached = client.caches.create(
    model="gemini-2.5-flash",
    config=types.CreateCachedContentConfig(
        contents=[
            types.Content(
                role="user",
                parts=[
                    types.Part.from_uri(file_uri=uris[0],
mime_type="application/pdf"),
                    types.Part.from_uri(file_uri=uris[1],
mime_type="application/pdf")
                ]
            )
        ],
        system_instruction="What is the sum of these documents?",
        display_name="test cache",
        ttl="3600s"
    )
)
print(cached.name)
```

This uploads content (possibly large documents) to the model's cache for faster inference [74].

- **Get Cached Content:**

```
cache_info = client.caches.get(name=cached.name)
```

- **List Caches:** Similar to other lists (`client.caches.list()`).

- **Delete Cache:**

```
client.caches.delete(name=cached.name)
```

- **Use Cache in Inference:** Pass `cached_content=<cache_name>` in `GenerateContentConfig` to use a cache:

```
resp = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Summarize the documents.",
```

```
    config=types.GenerateContentConfig(cached_content=cached.name)
)
print(resp.text)
```

This tells the model to use the pre-uploaded cache [75].

Tunings (`client.tunings`)

(Vertex AI only) The `client.tunings` service handles supervised fine-tuning jobs:

- **Create a Tuning Job:** Provide a base model and training data (as a GCS URI or `TuningDataset`):

```
from google.genai import types
model = "gemini-2.5-flash"
training = types.TuningDataset(gcs_uri="gs://bucket/training-data.jsonl")
job = client.tunings.tune(
    base_model=model,
    training_dataset=training,
    config=types.CreateTuningJobConfig(epoch_count=1,
tuned_model_display_name="finetuned-model")
)
print(job.name)
```

This starts a tuning job. The `job` object has a `.state` and ends with a `tuned_model` endpoint in its response [76].

- **Get Tuning Job:**

```
job = client.tunings.get(name=job.name)
print(job.state)
```

Poll until `job.state` is one of `JOB_STATE_SUCCEEDED/FAILED/CANCELLED` [77].

- **List Tuning Jobs:**

```
for job in client.tunings.list():
    print(job.name)
```

Use paging or async similarly to other lists [78].

- **Use Tuned Model:** Once complete, the `job.tuned_model.endpoint` is the new model resource. Use it for generation:

```
resp = client.models.generate_content(model=job.tuned_model.endpoint,
contents="Hello")
print(resp.text)
```

You can also get/update the tuned model itself via `client.models.get(model=job.tuned_model.model)` or `client.models.update(...)` [79] [80].

- **List Tuned Models:** (Base models excluded)

```
for model in client.models.list(config={'page_size': 10, 'query_base':
False}):
    print(model.name)
```

Set `query_base=False` to filter for tuned endpoints [81]. Async listing is available as well [82].

- **Update Tuned Model:** You can rename or describe a tuned model:

```
from google.genai import types
tuned = client.models.update(
    model=tuning_job.tuned_model.model,
    config=types.UpdateModelConfig(display_name="Updated name",
description="...")
)
```

or update by model name from listing [83].

Batch Predictions (`client.batches`)

The `client.batches` service supports large-scale (batch) requests:

- **Create Batch Job (Vertex AI):** Use a BigQuery or GCS source:

```
job = client.batches.create(
    model="gemini-2.5-flash",
    src="bq://my-project.my-dataset.my-table"
)
print(job.name)
```

This starts a batch job using a BQ table (or `gs://` file) of requests [84].

- **Create Batch Job (Gemini Dev API):** Pass an inlined list of requests:

```
batch_job = client.batches.create(
    model="gemini-2.5-flash",
    src=[{
        "contents": [{"parts":[{"text":"Hello!"}],"role":"user"}],
        "config": {"response_modalities": ["text"]}
    }]
)
```

Or upload a JSONL file of requests and reference it:

```
# Prepare myrequests.json with key/request per line
```

```
file_name = client.files.upload(file="myrequests.json",
config=types.UploadFileConfig(display_name="batch-data"))
batch_job = client.batches.create(
    model="gemini-2.0-flash",
    src=f"files/{file_name.name}"
)
```

(Use `UploadFileConfig` to name the file) [85] .

- **Get Batch Job:**

```
job = client.batches.get(name=job.name)
print(job.state)
```

Poll until `job.state` is in terminal states (`SUCCEEDED/FAILED/CANCELLED/PAUSED`) [86] .

- **List Batch Jobs:**

```
for job in
client.batches.list(config=types.ListBatchJobsConfig(page_size=10)):
    print(job.name)
```

Use paging (`pager = client.batches.list(...)` then `pager.next_page()`) and async versions similarly [87] [88] .

- **Delete Batch Job:**

```
client.batches.delete(name=job.name)
```

(Removes the job resource) [89].

Error Handling

The SDK throws a specific `genai.errors.APIError` for service errors. Example:

```
from google.genai import errors
try:
    client.models.generate_content(model="invalid-model", contents="Test")
except errors.APIError as e:
    print(e.code)    # HTTP error code (e.g. 404)
    print(e.message) # Error message string
```

This catches errors from the service [90].

Extra Features

- **Extra Request Body Fields:** You can use `HttpOptions.extra_body` to include arbitrary JSON in the request. This is for accessing undocumented or experimental API features. The extra dict must match the backend API schema. For example:

  ```
  response = client.models.generate_content(
      model="gemini-2.5-flash",
      contents="Hello",
      config=types.GenerateContentConfig(
          tools=[get_weather],
          http_options=types.HttpOptions(
              extra_body={'tool_config': {'function_calling_config': {'mode':
  'COMPOSITIONAL'}}}
          )
      )
  )
  ```

  (This `extra_body` merges into the request JSON sent to the server) [91].

References

For full class/method reference, see the SDK docs **Reference** section. Key classes include:

- `genai.Client`: Client configuration (properties: `api_key`, `vertexai`, `project`, `location`, `debug_config`, `http_options`) and service attributes (`client.models`, `client.chats`, `client.files`, `client.caches`, `client.tunings`, `client.batches`, etc.) [92] [93].
- `genai.AsyncClient`: Async counterpart (`aio` client).

- Service classes like `genai.Models` / `AsyncModels`, `genai.Chats` / `AsyncChats`, etc, with methods as shown above.
- Types in `google.genai.types`: content parts (`UserContent`, `ModelContent`), config classes (`GenerateContentConfig`, `EmbedContentConfig`, `ImageConfig`, etc), function/tool classes (`FunctionDeclaration`, `Tool`, `AutomaticFunctionCallingConfig`), tokenizer (`LocalTokenizer`), file upload config (`UploadFileConfig`), batch job configs, etc.

This guide has covered **all** public SDK features and examples. By following these patterns and using the examples above, a developer can fully leverage the Google Gen AI Python SDK for generative tasks [1] [90].

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] Google Gen AI SDK documentation
https://googleapis.github.io/python-genai/