Public

Developer's Guide to Gemini 3 Flash

On Google Cloud Vertex AI

Gemini 3 Flash represents the high-speed, cost-efficient engine of the Gemini 3 family, designed to enable the "agentic future" by removing latency and cost barriers that previously hindered high-volume AI deployment.

- High-Throughput Processing: Analyzes massive video/audio archives and extracts document data instantly.
- Cost-Efficient Scale: Enables sophisticated reasoning across high-volume processes where cost was previously a blocker.
- Low Latency: Powers real-time voice, chat, and decision-making where every second counts.

This guide provides a consolidated, hands-on path to get started with Gemini 3 Flash on Vertex AI, highlighting Gemini 3 Flash's key features and best practices.

Key resources

- Documentation: Get started with Gemini 3
- Colab intro_gemini_3_flash.ipynb
- Gemini 3 Flash Model Card

🚀 Quickstart

Before you begin, you must authenticate to Vertex AI using an API key or application default credentials (ADC). See authentication methods for more information.

Install the Google Gen AI SDK

Gemini 3 API features require Gen AI SDK for Python version `1.56.0` or later.

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

Set environment variables to use the Gen AI SDK with Vertex AI

Replace the `GOOGLE_CLOUD_PROJECT` value with your Google Cloud Project ID.
Gemini 3 Flash Preview model `gemini-3-flash-preview` is only available on **global** endpoints.

Public

```shell
export GOOGLE_CLOUD_PROJECT=GOOGLE_CLOUD_PROJECT
export GOOGLE_CLOUD_LOCATION=global
export GOOGLE_GENAI_USE_VERTEXAI=True
```

Make your first request

By default, Gemini 3 uses dynamic thinking to reason through prompts. For faster, lower-latency responses when complex reasoning isn't required, you can constrain the model's thinking with `thinking_level='minimal'` in Gemini 3 Flash. **Minimal** thinking level matches the "no thinking" setting for most queries. The model may think very minimally for complex tasks like coding. It minimizes latency for chat or high throughput applications.

Fast, low latency response options:

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

client = genai.Client()

response = client.models.generate_content(
   model="gemini-3-flash-preview",
   contents="How does AI work?",
   config=types.GenerateContentConfig(
       thinking_config=types.ThinkingConfig(
           thinking_level=types.ThinkingLevel.MINIMAL
       )
   ),
)
print(response.text)
```

🎉 New API features

Gemini 3 introduces powerful API enhancements and new parameters designed to give developers **granular control over performance (latency, cost), model behavior, and multimodal fidelity**.

This table summarizes the core new features and parameters available, along with direct links to their detailed documentation:

New Feature/API Change	Documentation
Model: `gemini-3-flash-preview`	Get started with Gemini 3 Model Card

Public

	Model Garden
Thinking Level	Thinking: Thinking Level API Reference
Media Resolution	Image understanding Video understanding Document understanding
Thought Signature	Thought signatures Thinking: Thought signature Function calling with thoughts
Temperature	Migration API Reference
Multimodal Function Responses	Function calling: Multimodal function responses
Streaming Function Calling	Function calling: Streaming Function Calling

1️⃣ Thinking Level

For Gemini 3 and beyond, we are simplifying how developers control the model's reasoning process. We are transitioning from the granular, integer-based `thinking_budget` to a streamlined, enum-based parameter `thinking_level`.

This change standardizes model behavior into distinct size options (minimal, low, medium, high), making it easier to predict model performance and manage latency.

Thinking Level	Gemini 3 Flash	Gemini 3 Pro	Description
`minimal`	Supported	Not supported	Matches the “no thinking” setting for most queries. The model may think very minimally for complex tasks like coding. Minimizes latency for chat or high throughput applications.
`low`	Supported	Supported	Light reasoning for most tasks with high speed and lower cost.
`medium`	Supported	Currently not supported	Balanced setting for most tasks requiring reasoning
`high`	Default (Dynamic)	Default (Dynamic)	Maximizes reasoning to optimize for task performance. The model may take longer than the medium setting to generate the first token, but the output will be more carefully reasoned.

The thinking_level parameter is optional. If omitted, the model will use its default setting (high).

Public

⚠️ **Notes:**

- You cannot use both `thinking_level` and the legacy `thinking_budget` parameter in the same request. Doing so will return a 400 error.
- If `thinking_level` is not specified, the model defaults to high, a dynamic setting that adjusts based on prompt complexity.
- For **Gemini 3 Flash**, while setting `thinking_level='minimal'` or `thinking_budget=0`, it does not guarantee the model will never think. In rare cases, the model may generate a small number of thinking tokens. Any thinking tokens generated under the minimal setting will be reflected in token count metadata and charged accordingly.
- **You must handle thought signature circulation even when** `thinking_level='minimal'`.

Gen AI SDK Example

Python

```python
prompt = """
Write a bash script that takes a matrix represented as a string with
format '[1,2],[3,4],[5,6]' and prints the transpose in the same format.
"""

response = client.models.generate_content(
   model="gemini-3-flash-preview",
   contents=prompt,
   config=types.GenerateContentConfig(
       thinking_config=types.ThinkingConfig(
           thinking_level=types.ThinkingLevel.HIGH  # Default, dynamic thinking
       )
   ),
)
print(response.text)
```

OpenAI Compatibility Example

For users utilizing the OpenAI compatibility layer, standard parameters are automatically mapped to Gemini 3 equivalents:

- `reasoning_effort` maps to `thinking_level`.
- `none`: maps to `thinking_level` minimal (Gemini 3 Flash only).
- `medium`: maps to `thinking_level` medium for Gemini 3 Flash, and `thinking_level` high for Gemini 3 Pro.

Public

```python
import openai
from google.auth import default
from google.auth.transport.requests import Request

credentials, _ = default(scopes=["https://www.googleapis.com/auth/cloud-platform"])

client = openai.OpenAI(

base_url=f"https://aiplatform.googleapis.com/v1/projects/{PROJECT_ID}/locations/global/endpoints/openapi",
    api_key=credentials.token,
)

prompt = """
Write a bash script that takes a matrix represented as a string with
format '[1,2],[3,4],[5,6]' and prints the transpose in the same format.
"""

response = client.chat.completions.create(
    model="gemini-3-flash-preview",
    reasoning_effort="medium",  # Map to thinking_level high.
    messages=[{"role": "user", "content": prompt}],
)

print(response.choices[0].message.content)
```

2️⃣ Media Resolution

The `media_resolution` parameter controls how the Gemini processes media inputs like images, videos, and PDF documents by determining the maximum number of tokens allocated for media inputs, allowing you to balance response quality against latency and cost.

You can configure media resolution in two ways:

- Per part
- Globally for an entire `generateContent` request (not available for `ultra_high`)

Available resolution values:

- `UNSPECIFIED (Default)`: The default setting. The token count for this level varies significantly between Gemini 3 and earlier Gemini models.
- `LOW`: Lower token count, resulting in faster processing and lower cost, but with less detail.
- `MEDIUM`: A balance between detail, cost, and latency.
- `HIGH`: Higher token count, providing more detail for the model to work with, at the expense of increased latency and cost.

Public

- `ULTRA_HIGH` (Per part and for image only): Highest token count, required for specific use cases such as computer use.

Note that **HIGH** provides the optimal performance for most use cases.

Token counts

The tables below summarize the approximate token counts for each media_resolution value and media type per model family.

Gemini 3 Models

Media Resolution	Image	Video	PDF
UNSPECIFIED (Default)	1120	70	560
LOW	280	70	280 + Text
MEDIUM	560	70	560 + Text
HIGH	1120	280	1120 + Text
ULTRA_HIGH	2240	N/A	N/A

Gemini 2.5 models

Media Resolution	Image	Video	PDF (Scanned)	PDF (Text)
UNSPECIFIED (Default)	256 + Pan & Scan (~2048)	256	256 + OCR	256 + Text
LOW	64	64	64 + OCR	64 + Text
MEDIUM	256	256	256 + OCR	256 + Text
HIGH	256 + Pan & Scan	256	256 + OCR	256 + Text

Recommended settings:

Media Resolution	Usage Guidance
`low`	Sufficient for most tasks.
`medium`	Document understanding (PDFs) including OCR and layout understanding.

Public

Media Resolution	Usage Guidance
`high`	Image analysis tasks to ensure maximum quality.
`ultra_high`	Computer use

Notes:

- **Token counts for multimodal inputs (images, video. Audio, PDF)** are an estimation based on the chosen `media_resolution`. As such, the result from the `count_tokens` API call may not match the final consumed tokens. The accurate usage for billing is only available after execution within the response's `usage_metadata`.
- Because `media_resolution` directly impacts token count, you may need to lower the resolution (e.g., to low) to fit very long inputs, such as long videos or extensive documents.

Example: set `media_resolution` per individual media part:

Python

```
response = client.models.generate_content(
  model="gemini-3-flash-preview",
  contents=[
      types.Part(
          file_data=types.FileData(
             file_uri="gs://cloud-samples-data/generative-ai/image/a-man-and-a-dog.png",
              mime_type="image/jpeg",
          ),
          media_resolution=types.PartMediaResolution(
              level=types.PartMediaResolutionLevel.MEDIA_RESOLUTION_ULTRA_HIGH  # Ultra
High resolution
          ),
      ),
      Part(
         file_data=types.FileData(

file_uri="gs://cloud-samples-data/generative-ai/video/behind_the_scenes_pixel.mp4",
            mime_type="video/mp4",
          ),
          media_resolution=types.PartMediaResolution(
              level=types.PartMediaResolutionLevel.MEDIA_RESOLUTION_LOW  # Low
resolution
          ),
      ),
      "When does the image appear in the video? What is the context?",
  ],
)
print(response.text)
```

Public

Example: Or set `media_resolution` globally (via `GenerateContentConfig`)

```python
response = client.models.generate_content(
  model="gemini-3-flash-preview",
  contents=[
      types.Part(
          file_data=types.FileData(

file_uri="gs://cloud-samples-data/generative-ai/image/a-man-and-a-dog.png",
              mime_type="image/jpeg",
          ),
      ),
      "What is in the image?",
  ],
  config=types.GenerateContentConfig(
      media_resolution=types.MediaResolution.MEDIA_RESOLUTION_LOW  # Global setting
  ),
)
print(response.text)
```

3 Thought Signatures

Thought signatures are encrypted tokens that preserve the model's reasoning state during multi-turn conversations, specifically when using Function Calling.

When a thinking model decides to call an external tool, it pauses its internal reasoning process. The thought signature acts as a "save state," allowing the model to resume its chain of thought seamlessly once you provide the function's result.

Why are thought signatures important?

Without thought signatures, the model "forgets" its specific reasoning steps during the tool execution phase. Passing the signature back ensures:

- **Context Continuity**: The model remembers why it called the tool.
- **Complex Reasoning**: Enables multi-step tasks where the output of one tool informs the reasoning for the next.

Where are thought signatures returned?

Gemini 3 enforces stricter validation and updated handling on thought signatures which were originally introduced in Gemini 2.5. To ensure the model maintains context across multiple turns of a conversation, you must return the thought signatures in your subsequent requests.

Public

- Model responses with a function call will always return a thought signature, regardless of the configured `thinking_level`.
- When there are parallel function calls, the first function call part returned by the model response will have a thought signature.
- When there are sequential function calls (multi-step), each function call will have a signature and clients are expected to pass signature back
- Model responses without a function call will return a thought signature inside the last part returned by the model.

How to handle thought signatures?

1) Automated Handling (Recommended)

If you are using the Google Gen AI SDKs (Python, Node.js, Go, Java) or OpenAI Chat Completions API, and utilizing the standard chat history features or appending the full model response, `thought_signatures` are handled automatically. You do not need to make any changes to your code.

Example: Manual Function Calling

When using the Gen AI SDK, thought signatures are handled automatically by appending the full model response in sequential model requests.

Python

```
# 1. Define your tool
get_weather_declaration = types.FunctionDeclaration(
   name="get_weather",
   description="Gets the current weather temperature for a given location.",
   parameters={
       "type": "object",
       "properties": {"location": {"type": "string"}},
       "required": ["location"],
   },
)
get_weather_tool = types.Tool(function_declarations=[get_weather_declaration])

# 2. Send a message that triggers the tool
prompt = "What's the weather like in London?"
response = client.models.generate_content(
   model="gemini-3-pro-preview",
   contents=prompt,
   config=types.GenerateContentConfig(
       tools=[get_weather_tool],
       thinking_config=types.ThinkingConfig(include_thoughts=True)
   ),
```

Public

```
)

# 4. Handle the function call
function_call = response.function_calls[0]
location = function_call.args["location"]
print(f"Model wants to call: {function_call.name}")

# Execute your tool (e.g., call an API)
# (This is a mock response for the example)
print(f"Calling external tool for: {location}")
function_response_data = {
   "location": location,
   "temperature": "30C",
}

# 5. Send the tool's result back
# Append this turn's messages to history for a final response.
# The `content` object automatically attaches the required thought_signature behind the
scenes.
history = [
   types.Content(role="user", parts=[types.Part(text=prompt)]),
   response.candidates[0].content,  # Signature preserved here
   types.Content(
     role="tool",
     parts=[
         types.Part.from_function_response(
             name=function_call.name,
             response=function_response_data,
         )
     ],
   )
]

response_2 = client.models.generate_content(
   model="gemini-3-flash-preview",
   contents=history,
   config=types.GenerateContentConfig(
       tools=[get_weather_tool],
       thinking_config=types.ThinkingConfig(include_thoughts=True)
   ),
)

# 6. Get the final, natural-language answer
print(f"\nFinal model response: {response_2.text}")
```

Public

Example: Automatic Function Calling

When using the Gen AI SDK in automatic function calling, thought signatures are handled automatically.

Python

```python
def get_current_temperature(location: str) -> dict:
    """Gets the current temperature for a given location.

    Args:
        location: The city and state, e.g. San Francisco, CA

    Returns:
        A dictionary containing the temperature and unit.
    """
    # ... (implementation) ...
    return {"temperature": 25, "unit": "Celsius"}

client = genai.Client()

response = client.models.generate_content(
    model="gemini-3-flash-preview",
    contents="What's the temperature in Boston?",
    config=types.GenerateContentConfig(
            tools=[get_current_temperature],
    )
)

print(response.text) # The SDK handles the function call and thought signature, and returns the final text
```

2) Manual Handling

If you are interacting with the API directly or managing raw JSON payloads, you must correctly handle the `thought_signature` included in the model's turn.

You **must** return this signature in the exact part where it was received when sending the conversation history back.

If proper signatures are not returned, Gemini 3 will return a 400 Error "`<Function Call>` in the <index of contents array> content block is missing a `thought_signature`".

Learn more about thought signatures in the documentation.

4 Multimodal Function Responses

Public

Multimodal function calling allows users to have function responses containing multimodal objects allowing for improved utilization of function calling capabilities of the model. Currently function calling only supports text based function responses.

Python

```
# This is a manual, two turn multimodal function calling workflow:

# 1. Define the function tool
get_image_declaration = types.FunctionDeclaration(
   name="get_image",
   description="Retrieves the image file reference for a specific order item.",
   parameters={
       "type": "object",
       "properties": {
           "item_name": {
               "type": "string",
               "description": "The name or description of the item ordered (e.g., 'green
shirt')."
           }
       },
       "required": ["item_name"],
   },
)
tool_config = types.Tool(function_declarations=[get_image_declaration])

# 2. Send a message that triggers the tool
prompt = "Show me the green shirt I ordered last month."
response_1 = client.models.generate_content(
   model="gemini-3-flash-preview",
   contents=[prompt],
   config=types.GenerateContentConfig(
       tools=[tool_config],
   )
)

# 3. Handle the function call
function_call = response_1.function_calls[0]
requested_item = function_call.args["item_name"]
print(f"Model wants to call: {function_call.name}")

# Execute your tool (e.g., call an API)
# (This is a mock response for the example)
print(f"Calling external tool for: {requested_item}")

function_response_data = {
  "image_ref": {"$ref": "dress.jpg"},
```

Public

```
}

function_response_multimodal_data = types.FunctionResponsePart(
   file_data=types.FunctionResponseFileData(
     mime_type="image/png",
     display_name="dress.jpg",
     file_uri="gs://cloud-samples-data/generative-ai/image/dress.jpg",
   )
)

# 4. Send the tool's result back
# Append this turn's messages to history for a final response.
history = [
  types.Content(role="user", parts=[types.Part(text=prompt)]),
  response_1.candidates[0].content,
  types.Content(
    role="tool",
    parts=[
        types.Part.from_function_response(
           name=function_call.name,
           response=function_response_data,
           parts=[function_response_multimodal_data]
        )
    ],
  )
]

response_2 = client.models.generate_content(
  model="gemini-3-flash-preview",
  contents=history,
  config=types.GenerateContentConfig(
      tools=[tool_config],
      thinking_config=types.ThinkingConfig(include_thoughts=True)
  ),
)

print(f"\nFinal model response: {response_2.text}")
```

5 Streaming Function Calling

You can use streaming partial function call arguments to improve streaming experience on tool use. This feature can be enabled by explicitly setting `stream_function_call_arguments` to `true`.

Public

```python
get_weather_declaration = types.FunctionDeclaration(
  name="get_weather",
  description="Gets the current weather temperature for a given location.",
  parameters={
      "type": "object",
      "properties": {"location": {"type": "string"}},
      "required": ["location"],
  },
)
get_weather_tool = types.Tool(function_declarations=[get_weather_declaration])


for chunk in client.models.generate_content_stream(
   model="gemini-3-flash-preview",
   contents="What's the weather in London and New York?",
   config=types.GenerateContentConfig(
       tools=[get_weather_tool],
       tool_config = types.ToolConfig(
           function_calling_config=types.FunctionCallingConfig(
               mode=types.FunctionCallingConfigMode.AUTO,
               stream_function_call_arguments=True,
           )
       ),
   ),
):
   function_call = chunk.function_calls[0]
   if function_call and function_call.name:
       print(f"{function_call.name}")
       print(f"will_continue={function_call.will_continue}")
```

Model response example:

```none
{
  "candidates": [
    {
      "content": {
        "role": "model",
        "parts": [
          {
            "functionCall": {
              "name": "get_weather",
              "willContinue": true
            }
  ...
```

Public

6️⃣ Temperature

- `Range for Gemini 3: 0.0 - 2.0 (default: 1.0)`

For Gemini 3, we strongly recommend keeping the `temperature` parameter at its default value of **`1.0`.**

While previous models often benefited from tuning temperature to control creativity versus determinism, Gemini 3's reasoning capabilities are optimized for the default setting.

Changing the temperature (setting it below `1.0`) may lead to unexpected behavior, such as looping or degraded performance, particularly in complex mathematical or reasoning tasks.

⭐ Supported Features

Gemini 3 Flash also supports the following features:

- System instructions
- Structured output
- Function calling
- Grounding with Google Search
- Code execution
- URL Context
- Thinking
- Context Caching
- Count Tokens
- Chat completions
- Batch prediction
- Provisioned Throughput
- Dynamic shared quota

Prompting Best Practices

Gemini 3 is a reasoning model, which changes how you should prompt.

- **Precise Instructions**: Be concise in your input prompts. Gemini 3 responds best to direct, clear instructions. It may over-analyze verbose or overly complex prompt engineering techniques used for older models.
- **Output Verbosity**: By default, Gemini 3 is less verbose and prefers providing direct, efficient answers. If your use case requires a more conversational or "chatty" persona, you must explicitly steer the model in the prompt (e.g., "Explain this as a friendly, talkative assistant").
- **Context Management**: When working with large datasets (e.g., entire books, codebases, or long videos), place your specific instructions or questions at the end of the prompt, after the data context. Anchor the model's reasoning to the provided data by starting your question with a phrase like, "Based on the information above...".

- **Temperature**: We strongly recommend keeping the `temperature` parameter at its default value of **`1.0`.** While previous models often benefited from tuning temperature to control creativity versus determinism, Gemini 3's reasoning capabilities are optimized for the default setting.
- **Grounding:** For grounding use cases, we recommend using the following developer instructions:

```
You are a strictly grounded assistant limited to the
information provided in the User Context. In your answers,
rely **only** on the facts that are directly mentioned in
that context. You must **not** access or utilize your own
knowledge or common sense to answer. Do not assume or
infer from the provided facts; simply report them exactly
as they appear. Your answer must be factual and fully
truthful to the provided text, leaving absolutely no room
for speculation or interpretation. Treat the provided
context as the absolute limit of truth; any facts or
details that are not directly mentioned in the context
must be considered **completely untruthful** and
**completely unsupported**. If the exact answer is not
explicitly written in the context, you must state that the
information is not available.
```

- **Google Search:** When using the Google Search tool, Gemini 3 Flash could at times confuse current date / time for events in 2024. This can result in the model formulating search queries for the wrong year.

  Mitigation: To ensure the model utilizes the correct time frame, explicitly reinforce the current date in the system instructions.

  Recommended system Instruction:

```
For time-sensitive user queries that require up-to-date
information, you MUST follow the provided current time
(date and year) when formulating search queries in tool
calls. Remember it is 2025 this year
```

- **Knowledge cutoff:** For certain queries, Gemini 3 Flash benefits from being explicitly told its knowledge cutoff . This is the case when the Google Search tool is disabled and the query explicitly requires the model to be able to identify the cutoff data in parametric knowledge.

  Recommendation: Adding the following clause to the system instruction:

```
Your knowledge cutoff date is January 2025.
```

- **Recommendation:** Use the Media Resolution parameter to control the maximum number of tokens the model uses to represent an image or frames in videos. High resolution will enable the model to capture details in an image, and may use more tokens per frame, while lower resolution allows for optimizing cost/latency for images with less visual detail.

- **Recommendation:** Use a higher Frame Per Second (FPS) sampling rate for videos requiring granular temporal analysis, such as fast-action understanding or high-speed motion tracking.

Public

Migration Considerations

When migrating, consider the following:

- **Thinking**: Gemini 3 and later models use the `thinking_level` parameter instead of `thinking_budget`.
- **Temperature Settings:** If your existing code explicitly sets `temperature` (especially to low values for deterministic outputs), we recommend removing this parameter and using the Gemini 3 default of `1.0` to avoid potential looping issues or performance degradation on complex tasks.
- **Thought Signatures**: For Gemini 3 and later models, if a thought signature is expected in a turn but not provided, the model returns an error instead of a warning.
- **Media Resolution and Tokenizatio**n: Gemini 3 and later models use a variable sequence length for media tokenization instead of Pan and Scan, and have new default resolutions and token costs for images, PDFs, and video.
- **Token Counting for Multimodality Input:** Token counts for multimodal inputs (images, video. audio) are an estimation based on the chosen `media_resolution`. As such, the result from the `count_tokens` API call may not match the final consumed tokens. The accurate usage for billing is only available after execution within the response's `usage_metadata`.
- **Token Consumption:** Migrating to Gemini 3 defaults may **increase** token usage for PDFs but **decrease** token usage for video. If requests now exceed the context window due to higher default resolutions, we recommend explicitly reducing the media resolution.
- **PDF & Document Understanding:** Default OCR resolution for PDFs has changed. If you relied on specific behavior for dense document parsing, test the new `media_resolution: "high"` setting to ensure continued accuracy. For Gemini 3 and later models, PDF token counts in usage_metadata are reported under the IMAGE modality instead of DOCUMENT.
- **PDF Processing**: For Gemini 3 and later models, OCR is not used by default when processing scanned PDFs.
- **Image Segmentation:** Image segmentation is not supported by Gemini 3 and later models. For workloads requiring native image segmentation, we recommend continuing to utilize Gemini 2.5 Flash with thinking turned off.
- **Multimodal Function Responses**: For Gemini 3 and later models, you can include image and PDF data in function responses.
- A key introduction for **Gemini 3 Flash** is the **`thinking_level="minimal"`** setting. This option is designed to replicate the behavior of **`thinking_budget=0`** from previous models (like 2.5 Flash), prioritizing speed and concise output.
  Notes:
  - **Behavior**: While setting `thinking_budget=0`, it does not guarantee the model will never think. In rare cases, the model may generate a small number of thinking tokens.
  - **Billing**: Any thinking tokens generated under the minimal setting will be reflected in token count metadata and charged accordingly.
  - **Availability**: This setting is currently supported on Gemini 3 Flash. It is not supported on Gemini 3.0 Pro.

Resources

- Documentation: Get started with Gemini 3

Public

- Colab intro_gemini_3_flash.ipynb
- Model Card