Context caching

✓ Python JavaScript Go REST

In a typical AI workflow, you might pass the same input tokens over and over to a model. The Gemini API offers two different caching mechanisms:

- Implicit caching (automatic, no cost saving guarantee)
- Explicit caching (manual, cost saving guarantee)

Implicit caching is enabled on Gemini 2.5 models by default. If a request contains content that is a cache hit, we automatically pass the cost savings back to you.

Explicit caching is useful in cases where you want to guarantee cost savings, but with some added developer work.

There are two layers of caches in Gemini:

- Prefill cache: Cache for models being ready to generate the first token. It cannot share between models.
- Preprocess cache: Cache for formatting and tokenizing multi-modal input like videos and PDF. It could be reused by different models

Implicit caching

Implicit caching is enabled by default for all Gemini 2.5 models. We automatically pass on cost savings if your request hits caches. There is nothing you need to do in order to enable this. It is effective as of May 8th, 2025. The minimum input token count for context caching is 1,024 for 2.5 Flash and 2,048 for 2.5 Pro.

To increase the chance of an implicit cache hit:

- Try putting large and common contents at the beginning of your prompt
- Try to send requests with similar prefix in a short amount of time

You can see the number of tokens which were cache hits in the response object's `usage_metadata` field.

The cost savings are measure by prefilled cache hits. Only prefilled cache and YouTube video preprocessing cache are enabled for implicit caching. For lower latency Gemini calls for other multi-modal input, consider using explicit cache.

Explicit caching

Using the Gemini API explicit caching feature, you can pass some content to the model once, cache the input tokens, and then refer to the cached tokens for subsequent requests. At certain volumes, using cached tokens is lower cost than passing in the same corpus of tokens repeatedly.

When you cache a set of tokens, you can choose how long you want the cache to exist before the tokens are automatically deleted. This caching duration is called the *time to live* (TTL). If not set, the TTL defaults to 1 hour. The cost for caching depends on the input token size and how long you want the tokens to persist.

This section assumes that you've installed a Gemini SDK (or have curl installed) and that you've configured an API key, as shown in the quickstart (/gemini-api/docs/quickstart).

Generate content using a cache

The following example shows how to generate content using a cached system instruction and video file.

VideosPDFs (#pdfs)
(#videos)

```
import os
import pathlib
import requests
import time

from google import genai
from google.genai import types

client = genai.Client()
```

```
# Download video file
url = 'https://storage.googleapis.com/generativeai-downloads/data/Sherlock
path_to_video_file = pathlib.Path('SherlockJr._10min.mp4')
if not path_to_video_file.exists():
  with path_to_video_file.open('wb') as wf:
    response = requests.get(url, stream=True)
    for chunk in response.iter_content(chunk_size=32768):
      wf.write(chunk)

# Upload the video using the Files API
video_file = client.files.upload(file=path_to_video_file)

# Wait for the file to finish processing
while video_file.state.name == 'PROCESSING':
  print('Waiting for video to be processed.')
  time.sleep(2)
  video_file = client.files.get(name=video_file.name)

print(f'Video processing complete: {video_file.uri}')

# You must use an explicit version suffix: "-flash-001", not just "-flash"
model='models/gemini-2.0-flash-001'

# Create a cache with a 5 minute TTL
cache = client.caches.create(
    model=model,
    config=types.CreateCachedContentConfig(
      display_name='sherlock jr movie', # used to identify the cache
      system_instruction=(
          'You are an expert video analyzer, and your job is to answer '
          'the user\'s query based on the video file you have access to.'
      ),
      contents=[video_file],
      ttl="300s",
  )
)

# Construct a GenerativeModel which uses the created cache.
response = client.models.generate_content(
  model = model,
  contents= (
    'Introduce different characters in the movie by describing '
    'their personality, looks, and names. Also list the timestamps '
    'they were introduced for the first time.'),
  config=types.GenerateContentConfig(cached_content=cache.name)
)

print(response.usage_metadata)

# The output should look something like this:
#
# prompt_token_count: 696219
# cached_content_token_count: 696190
# candidates_token_count: 214
# total_token_count: 696433

print(response.text)
```

List caches

It's not possible to retrieve or view cached content, but you can retrieve cache metadata (**`name`**, **`model`**, **`display_name`**, **`usage_metadata`**, **`create_time`**, **`update_time`**, and **`expire_time`**).

To list metadata for all uploaded caches, use **`CachedContent.list()`**:

```
for cache in client.caches.list():
  print(cache)
```

To fetch the metadata for one cache object, if you know its name, use **`get`**:

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

Update a cache

You can set a new **`ttl`** or **`expire_time`** for a cache. Changing anything else about the cache isn't supported.

The following example shows how to update the **`ttl`** of a cache using **`client.caches.update()`**.

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

client.caches.update(
  name = cache.name,
  config  = types.UpdateCachedContentConfig(
      ttl='300s'
  )
)
```

To set the expiry time, it will accepts either a `datetime` object or an ISO-formatted datetime string (`dt.isoformat()`, like `2025-01-27T16:02:36.473528+00:00`). Your time must include a time zone (`datetime.utcnow()` doesn't attach a time zone, `datetime.now(datetime.timezone.utc)` does attach a time zone).

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

# You must use a time zone-aware time.
in10min = datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(minu

client.caches.update(
  name = cache.name,
  config  = types.UpdateCachedContentConfig(
      expire_time=in10min
  )
)
```

Delete a cache

The caching service provides a delete operation for manually removing content from the cache. The following example shows how to delete a cache:

```
client.caches.delete(cache.name)
```

Explicit caching using the OpenAI library

If you're using an OpenAI library (/gemini-api/docs/openai), you can enable explicit caching using the `cached_content` property on `extra_body` (/gemini-api/docs/openai#extra-body).

When to use explicit caching

Context caching is particularly well suited to scenarios where a substantial initial context is referenced repeatedly by shorter requests. Consider using context caching for use cases such as:

- Chatbots with extensive system instructions (/gemini-api/docs/system-instructions)
- Repetitive analysis of lengthy video files
- Recurring queries against large document sets
- Frequent code repository analysis or bug fixing

How explicit caching reduces costs

Context caching is a paid feature designed to reduce overall operational costs. Billing is based on the following factors:

1. **Cache token count:** The number of input tokens cached, billed at a reduced rate when included in subsequent prompts.
2. **Storage duration:** The amount of time cached tokens are stored (TTL), billed based on the TTL duration of cached token count. There are no minimum or maximum bounds on the TTL.
3. **Other factors:** Other charges apply, such as for non-cached input tokens and output tokens.

For up-to-date pricing details, refer to the Gemini API pricing page (/pricing). To learn how to count tokens, see the Token guide (/gemini-api/docs/tokens).

Additional considerations

Keep the following considerations in mind when using context caching:

- The *minimum* input token count for context caching is 1,024 for 2.5 Flash and 2,048 for

2.5 Pro. The *maximum* is the same as the maximum for the given model. (For more on counting tokens, see the Token guide (/gemini-api/docs/tokens)).

- The model doesn't make any distinction between cached tokens and regular input tokens. Cached content is a prefix to the prompt.
- There are no special rate or usage limits on context caching; the standard rate limits for `GenerateContent` apply, and token limits include cached tokens.
- The number of cached tokens is returned in the `usage_metadata` from the create, get, and list operations of the cache service, and also in `GenerateContent` when using the cache.

Last updated 2025-06-03 UTC.