# Prompt caching

Copy page

Prompt caching is a powerful feature that optimizes your API usage by allowing resuming from specific prefixes in your prompts. This approach significantly reduces processing time and costs for repetitive tasks or prompts with consistent elements.

Here's an example of how to implement prompt caching with the Messages API using a `cache_control` block:

Shell

```
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "claude-sonnet-4-5",
    "max_tokens": 1024,
    "system": [
      {
        "type": "text",
        "text": "You are an AI assistant tasked with analyzing literary works.
      },
      {
        "type": "text",
        "text": "<the entire contents of Pride and Prejudice>",
        "cache_control": {"type": "ephemeral"}
      }
    ],
    "messages": [
      {
        "role": "user",
        "content": "Analyze the major themes in Pride and Prejudice."
      }
    ]
  }'

# Call the model again with the same inputs up to the cache checkpoint
curl https://api.anthropic.com/v1/messages # rest of input
```

JSON

```json
{"cache_creation_input_tokens":188086,"cache_read_input_tokens":0,"input_token
{"cache_creation_input_tokens":0,"cache_read_input_tokens":188086,"input_token
```

In this example, the entire text of "Pride and Prejudice" is cached using the `cache_control` parameter. This enables reuse of this large text across multiple API calls without reprocessing it each time. Changing only the user message allows you to ask various questions about the book while utilizing the cached content, leading to faster responses and improved efficiency.

When you send a request with prompt caching enabled:

1. The system checks if a prompt prefix, up to a specified cache breakpoint, is already cached from a recent query.
2. If found, it uses the cached version, reducing processing time and costs.
3. Otherwise, it processes the full prompt and caches the prefix once the response begins.

This is especially useful for:

- Prompts with many examples
- Large amounts of context or background information
- Repetitive tasks with consistent instructions
- Long multi-turn conversations

By default, the cache has a 5-minute lifetime. The cache is refreshed for no additional cost each time the cached content is used.

If you find that 5 minutes is too short, Anthropic also offers a 1-hour cache duration at additional cost.

For more information, see 1-hour cache duration.

**Prompt caching caches the full prefix**

Prompt caching references the entire prompt - `tools` , `system` , and `messages` (in that order) up to and including the block designated with `cache_control` .

## Pricing

Prompt caching introduces a new pricing structure. The table below shows the price per million tokens for each supported model:

| | | | | | |
|---|---|---|---|---|---|
| Claude Opus 4.5 | $5 / MTok | $6.25 / MTok | $10 / MTok | $0.50 / MTok | $25 / MTok |
| Claude Opus 4.1 | $15 / MTok | $18.75 / MTok | $30 / MTok | $1.50 / MTok | $75 / MTok |
| Claude Opus 4 | $15 / MTok | $18.75 / MTok | $30 / MTok | $1.50 / MTok | $75 / MTok |
| Claude Sonnet 4.5 | $3 / MTok | $3.75 / MTok | $6 / MTok | $0.30 / MTok | $15 / MTok |
| Claude Sonnet 4 | $3 / MTok | $3.75 / MTok | $6 / MTok | $0.30 / MTok | $15 / MTok |
| Claude Sonnet 3.7 (deprecated) | $3 / MTok | $3.75 / MTok | $6 / MTok | $0.30 / MTok | $15 / MTok |
| Claude Haiku 4.5 | $1 / MTok | $1.25 / MTok | $2 / MTok | $0.10 / MTok | $5 / MTok |
| Claude Haiku 3.5 | $0.80 / MTok | $1 / MTok | $1.6 / MTok | $0.08 / MTok | $4 / MTok |
| Claude Opus 3 (deprecated) | $15 / MTok | $18.75 / MTok | $30 / MTok | $1.50 / MTok | $75 / MTok |
| Claude Haiku 3 | $0.25 / MTok | $0.30 / MTok | $0.50 / MTok | $0.03 / MTok | $1.25 / MTok |

The table above reflects the following pricing multipliers for prompt caching:

- 5-minute cache write tokens are 1.25 times the base input tokens price
- 1-hour cache write tokens are 2 times the base input tokens price
- Cache read tokens are 0.1 times the base input tokens price

## How to implement prompt caching

### Supported models

Prompt caching is currently supported on:

- Claude Opus 4.5
- Claude Opus 4.1
- Claude Opus 4

- Claude Sonnet 3.7 (deprecated)
- Claude Haiku 4.5
- Claude Haiku 3.5 (deprecated)
- Claude Haiku 3

## Structuring your prompt

Place static content (tool definitions, system instructions, context, examples) at the beginning of your prompt. Mark the end of the reusable content for caching using the `cache_control` parameter.

Cache prefixes are created in the following order: `tools`, `system`, then `messages`. This order forms a hierarchy where each level builds upon the previous ones.

### How automatic prefix checking works

You can use just one cache breakpoint at the end of your static content, and the system will automatically find the longest matching sequence of cached blocks. Understanding how this works helps you optimize your caching strategy.

**Three core principles:**

1. **Cache keys are cumulative**: When you explicitly cache a block with `cache_control`, the cache hash key is generated by hashing all previous blocks in the conversation sequentially. This means the cache for each block depends on all content that came before it.
2. **Backward sequential checking**: The system checks for cache hits by working backwards from your explicit breakpoint, checking each previous block in reverse order. This ensures you get the longest possible cache hit.
3. **20-block lookback window**: The system only checks up to 20 blocks before each explicit `cache_control` breakpoint. After checking 20 blocks without a match, it stops checking and moves to the next explicit breakpoint (if any).

**Example: Understanding the lookback window**

Consider a conversation with 30 content blocks where you set `cache_control` only on block 30:

- **If you send block 31 with no changes to previous blocks**: The system checks block 30 (match!). You get a cache hit at block 30, and only block 31 needs processing.

a cache hit at block 24, and only blocks 25-30 need reprocessing.

- **If you modify block 5 and send block 31**: The system checks backwards from block 30 → 29 → 28... → 11 (check #20). After 20 checks without finding a match, it stops looking. Since block 5 is beyond the 20-block window, no cache hit occurs and all blocks need reprocessing. However, if you had set an explicit `cache_control` breakpoint on block 5, the system would continue checking from that breakpoint: block 5 (no match) → block 4 (match!). This allows a cache hit at block 4, demonstrating why you should place breakpoints before editable content.

**Key takeaway**: Always set an explicit cache breakpoint at the end of your conversation to maximize your chances of cache hits. Additionally, set breakpoints just before content blocks that might be editable to ensure those sections can be cached independently.

### When to use multiple breakpoints

You can define up to 4 cache breakpoints if you want to:

- Cache different sections that change at different frequencies (e.g., tools rarely change, but context updates daily)
- Have more control over exactly what gets cached
- Ensure caching for content more than 20 blocks before your final breakpoint
- Place breakpoints before editable content to guarantee cache hits even when changes occur beyond the 20-block window

**Important limitation:** If your prompt has more than 20 content blocks before your cache breakpoint, and you modify content earlier than those 20 blocks, you won't get a cache hit unless you add additional explicit breakpoints closer to that content.

## Cache limitations

The minimum cacheable prompt length is:

- 4096 tokens for Claude Opus 4.5
- 1024 tokens for Claude Opus 4.1, Claude Opus 4, Claude Sonnet 4.5, Claude Sonnet 4, and Claude Sonnet 3.7 (deprecated)
- 4096 tokens for Claude Haiku 4.5
- 2048 tokens for Claude Haiku 3.5 (deprecated) and Claude Haiku 3

prompt was cached, see the response usage fields.

For concurrent requests, note that a cache entry only becomes available after the first response begins. If you need cache hits for parallel requests, wait for the first response before sending subsequent requests.

Currently, "ephemeral" is the only supported cache type, which by default has a 5-minute lifetime.

## Understanding cache breakpoint costs

**Cache breakpoints themselves don't add any cost.** You are only charged for:

- **Cache writes**: When new content is written to the cache (25% more than base input tokens for 5-minute TTL)
- **Cache reads**: When cached content is used (10% of base input token price)
- **Regular input tokens**: For any uncached content

Adding more `cache_control` breakpoints doesn't increase your costs - you still pay the same amount based on what content is actually cached and read. The breakpoints simply give you control over what sections can be cached independently.

## What can be cached

Most blocks in the request can be designated for caching with `cache_control`. This includes:

- Tools: Tool definitions in the `tools` array
- System messages: Content blocks in the `system` array
- Text messages: Content blocks in the `messages.content` array, for both user and assistant turns
- Images & Documents: Content blocks in the `messages.content` array, in user turns
- Tool use and tool results: Content blocks in the `messages.content` array, in both user and assistant turns

Each of these elements can be marked with `cache_control` to enable caching for that portion of the request.

## What cannot be cached

While most request blocks can be cached, there are some exceptions:

assistant turns. When cached this way, they DO count as input tokens when read from cache.

- Sub-content blocks (like citations) themselves cannot be cached directly. Instead, cache the top-level block.

  In the case of citations, the top-level document content blocks that serve as the source material for citations can be cached. This allows you to use prompt caching with citations effectively by caching the documents that citations will reference.

- Empty text blocks cannot be cached.

## What invalidates the cache

Modifications to cached content can invalidate some or all of the cache.

As described in Structuring your prompt, the cache follows the hierarchy: `tools` → `system` → `messages`. Changes at each level invalidate that level and all subsequent levels.

The following table shows which parts of the cache are invalidated by different types of changes. ✘ indicates that the cache is invalidated, while ✓ indicates that the cache remains valid.

| What changes | Tools cache | System cache | Messages cache | Impact |
|---|---|---|---|---|
| **Tool definitions** | ✘ | ✘ | ✘ | Modifying tool definitions (names, descriptions, parameters) invalidates the entire cache |
| **Web search toggle** | ✓ | ✘ | ✘ | Enabling/disabling web search modifies the system prompt |
| **Citations toggle** | ✓ | ✘ | ✘ | Enabling/disabling citations modifies the system prompt |
| **Tool choice** | ✓ | ✓ | ✘ | Changes to `tool_choice` parameter only affect message blocks |
| **Images** | ✓ | ✓ | ✘ | Adding/removing images anywhere in the prompt affects message blocks |
| **Thinking parameters** | ✓ | ✓ | ✘ | Changes to extended thinking settings (enable/disable, budget) affect message blocks |
| **Non-tool results passed to extended** | ✓ | ✓ | ✘ | When non-tool results are passed in requests while extended thinking is enabled, all previously-cached thinking blocks are stripped |

| | |
|---|---|
| **thinking requests** | from context, and any messages in context that follow those thinking blocks are removed from the cache. For more details, see Caching with thinking blocks. |

## Tracking cache performance

Monitor cache performance using these API response fields, within `usage` in the response (or `message_start` event if streaming):

- `cache_creation_input_tokens` : Number of tokens written to the cache when creating a new entry.
- `cache_read_input_tokens` : Number of tokens retrieved from the cache for this request.
- `input_tokens` : Number of input tokens which were not read from or used to create a cache (i.e., tokens after the last cache breakpoint).

**Understanding the token breakdown**

The `input_tokens` field represents only the tokens that come **after the last cache breakpoint** in your request - not all the input tokens you sent.

To calculate total input tokens:

```
total_input_tokens = cache_read_input_tokens + cache_creation_inp to
```

**Spatial explanation:**

- `cache_read_input_tokens` = tokens before breakpoint already cached (reads)
- `cache_creation_input_tokens` = tokens before breakpoint being cached now (writes)
- `input_tokens` = tokens after your last breakpoint (not eligible for cache)

**Example:** If you have a request with 100,000 tokens of cached content (read from cache), 0 tokens of new content being cached, and 50 tokens in your user message (after the cache breakpoint):

- `cache_read_input_tokens` : 100,000
- `cache_creation_input_tokens` : 0

This is important for understanding both costs and rate limits, as `input_tokens` will typically be much smaller than your total input when using caching effectively.

## Best practices for effective caching

To optimize prompt caching performance:

- Cache stable, reusable content like system instructions, background information, large contexts, or frequent tool definitions.
- Place cached content at the prompt's beginning for best performance.
- Use cache breakpoints strategically to separate different cacheable prefix sections.
- Set cache breakpoints at the end of conversations and just before editable content to maximize cache hit rates, especially when working with prompts that have more than 20 content blocks.
- Regularly analyze cache hit rates and adjust your strategy as needed.

## Optimizing for different use cases

Tailor your prompt caching strategy to your scenario:

- Conversational agents: Reduce cost and latency for extended conversations, especially those with long instructions or uploaded documents.
- Coding assistants: Improve autocomplete and codebase Q&A by keeping relevant sections or a summarized version of the codebase in the prompt.
- Large document processing: Incorporate complete long-form material including images in your prompt without increasing response latency.
- Detailed instruction sets: Share extensive lists of instructions, procedures, and examples to fine-tune Claude's responses. Developers often include an example or two in the prompt, but with prompt caching you can get even better performance by including 20+ diverse examples of high quality answers.
- Agentic tool use: Enhance performance for scenarios involving multiple tool calls and iterative code changes, where each step typically requires a new API call.
- Talk to books, papers, documentation, podcast transcripts, and other longform content: Bring any knowledge base alive by embedding the entire document(s) into the prompt, and letting users ask it questions.

If experiencing unexpected behavior:

- Ensure cached sections are identical and marked with cache_control in the same locations across calls
- Check that calls are made within the cache lifetime (5 minutes by default)
- Verify that `tool_choice` and image usage remain consistent between calls
- Validate that you are caching at least the minimum number of tokens
- The system automatically checks for cache hits at previous content block boundaries (up to ~20 blocks before your breakpoint). For prompts with more than 20 content blocks, you may need additional `cache_control` parameters earlier in the prompt to ensure all content can be cached
- Verify that the keys in your `tool_use` content blocks have stable ordering as some languages (e.g. Swift, Go) randomize key order during JSON conversion, breaking caches

Changes to `tool_choice` or the presence/absence of images anywhere in the prompt will invalidate the cache, requiring a new cache entry to be created. For more details on cache invalidation, see What invalidates the cache.

## Caching with thinking blocks

When using extended thinking with prompt caching, thinking blocks have special behavior:

**Automatic caching alongside other content**: While thinking blocks cannot be explicitly marked with `cache_control`, they get cached as part of the request content when you make subsequent API calls with tool results. This commonly happens during tool use when you pass thinking blocks back to continue the conversation.

**Input token counting**: When thinking blocks are read from cache, they count as input tokens in your usage metrics. This is important for cost calculation and token budgeting.

**Cache invalidation patterns**:

- Cache remains valid when only tool results are provided as user messages
- Cache gets invalidated when non-tool-result user content is added, causing all previous thinking blocks to be stripped
- This caching behavior occurs even without explicit `cache_control` markers

```
Request 1: User: "What's the weather in Paris?"
Response: [thinking_block_1] + [tool_use block 1]

Request 2:
User: ["What's the weather in Paris?"],
Assistant: [thinking_block_1] + [tool_use block 1],
User: [tool_result_1, cache=True]
Response: [thinking_block_2] + [text block 2]
# Request 2 caches its request content (not the response)
# The cache includes: user message, thinking_block_1, tool_use block 1, and to

Request 3:
User: ["What's the weather in Paris?"],
Assistant: [thinking_block_1] + [tool_use block 1],
User: [tool_result_1, cache=True],
Assistant: [thinking_block_2] + [text block 2],
User: [Text response, cache=True]
# Non-tool-result user block causes all thinking blocks to be ignored
# This request is processed as if thinking blocks were never present
```

When a non-tool-result user block is included, it designates a new assistant loop and all previous thinking blocks are removed from context.

For more detailed information, see the extended thinking documentation.

## Cache storage and sharing

- **Organization Isolation**: Caches are isolated between organizations. Different organizations never share caches, even if they use identical prompts.
- **Exact Matching**: Cache hits require 100% identical prompt segments, including all text and images up to and including the block marked with cache control.
- **Output Token Generation**: Prompt caching has no effect on output token generation. The response you receive will be identical to what you would get if prompt caching was not used.

Claude Docs

If you find that 5 minutes is too short, Anthropic also offers a 1-hour cache duration at additional cost.

To use the extended cache, include `ttl` in the `cache_control` definition like this:

```
"cache_control": {
    "type": "ephemeral",
    "ttl": "5m" | "1h"
}
```

The response will include detailed cache information like the following:

```
{
    "usage": {
        "input_tokens": ...,
        "cache_read_input_tokens": ...,
        "cache_creation_input_tokens": ...,
        "output_tokens": ...,

        "cache_creation": {
            "ephemeral_5m_input_tokens": 456,
            "ephemeral_1h_input_tokens": 100,
        }
    }
}
```

Note that the current `cache_creation_input_tokens` field equals the sum of the values in the `cache_creation` object.

## When to use the 1-hour cache

If you have prompts that are used at a regular cadence (i.e., system prompts that are used more frequently than every 5 minutes), continue to use the 5-minute cache, since this will continue to be refreshed at no additional charge.

The 1-hour cache is best used in the following scenarios:

- When you have prompts that are likely used less frequently than 5 minutes, but more frequently than every hour. For example, when an agentic side-agent will take longer than 5 minutes, or when storing a long chat conversation with a user and you generally expect that user may not respond in the next 5 minutes.

- When you want to improve your rate limit utilization, since cache hits are not deducted against your rate limit.

> The 5-minute and 1-hour cache behave the same with respect to latency. You will generally see improved time-to-first-token for long documents.

## Mixing different TTLs

You can use both 1-hour and 5-minute cache controls in the same request, but with an important constraint: Cache entries with longer TTL must appear before shorter TTLs (i.e., a 1-hour cache entry must appear before any 5-minute cache entries).

When mixing TTLs, we determine three billing locations in your prompt:

1. Position `A` : The token count at the highest cache hit (or 0 if no hits).
2. Position `B` : The token count at the highest 1-hour `cache_control` block after `A` (or equals `A` if none exist).
3. Position `C` : The token count at the last `cache_control` block.

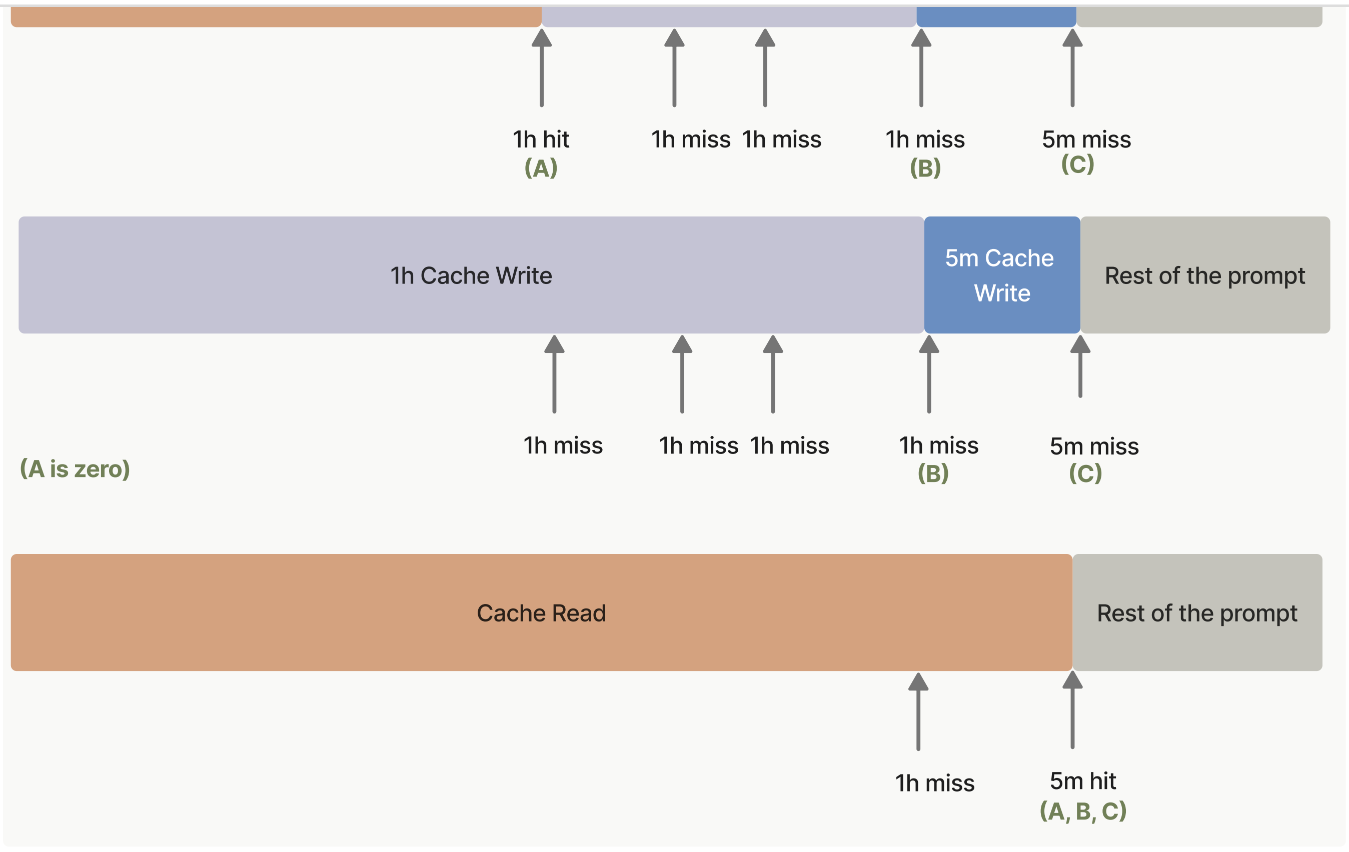
> If `B` and/or `C` are larger than `A` , they will necessarily be cache misses, because `A` is the highest cache hit.

You'll be charged for:

1. Cache read tokens for `A` .
2. 1-hour cache write tokens for `(B - A)` .
3. 5-minute cache write tokens for `(C - B)` .

Here are 3 examples. This depicts the input tokens of 3 requests, each of which has different cache hits and cache misses. Each has a different calculated pricing, shown in the colored boxes, as a result.

Claude Docs



## Prompt caching examples

To help you get started with prompt caching, we've prepared a prompt caching cookbook with detailed examples and best practices.

Below, we've included several code snippets that showcase various prompt caching patterns. These examples demonstrate how to implement caching in different scenarios, helping you understand the practical applications of this feature:

> Large context caching example

> Caching tool definitions

> Continuing a multi-turn conversation

> Putting it all together: Multiple cache breakpoints

Claude Docs

## FAQ

- Do I need multiple cache breakpoints or is one at the end sufficient?
- Do cache breakpoints add extra cost?
- How do I calculate total input tokens from the usage fields?
- What is the cache lifetime?
- How many cache breakpoints can I use?
- Is prompt caching available for all models?
- How does prompt caching work with extended thinking?
- How do I enable prompt caching?
- Can I use prompt caching with other API features?
- How does prompt caching affect pricing?
- Can I manually clear the cache?
- How can I track the effectiveness of my caching strategy?
- What can break the cache?
- How does prompt caching handle privacy and data separation?
- Can I use prompt caching with the Batches API?

Claude Docs

Why am I seeing 'TypeError: Cannot read properties of undefined (reading 'messages')'?

Claude Docs

Solutions

AI agents

Code modernization

Coding

Customer support

Education

Financial services

Government

Life sciences

Partners

Amazon Bedrock

Google Cloud's Vertex AI

Learn

Blog

Catalog

Courses

Use cases

Connectors

Customer stories

Engineering at Anthropic

Events

Powered by Claude

Service partners

Startups program

Company

Anthropic

Careers

Economic Futures

Research

News

Responsible Scaling Policy

Security and compliance

Transparency

Help and security

Claude Docs

Support

Discord

Terms and policies

Privacy policy

Responsible disclosure policy

Terms of service: Commercial

Terms of service: Consumer

Usage policy