

Mastering Anthropic's Claude: A Comprehensive Guide

This book serves as an essential guide to mastering Anthropic's Claude AI platform. From introductory concepts to advanced functionalities, discover how to effectively utilize Claude's capabilities in language, reasoning, coding, and more. Ideal for developers and enthusiasts looking to harness Claude's potential.

Table of Contents

First Steps

- [Intro to Claude](#)
- [Get started](#)

Models & Pricing

- [Models overview](#)
- [Choosing a model](#)
- [Migrating to Claude 4](#)
- [Model deprecations](#)
- [Pricing](#)

Learn About Claude

- [Building with Claude](#)
- [Features overview](#)
- [Context windows](#)
- [Glossary](#)

Capabilities

- [Prompt caching](#)
- [Extended thinking](#)
- [Streaming Messages](#)
- [Batch processing](#)
- [Citations](#)
- [Multilingual support](#)
- [Token counting](#)
- [Embeddings](#)
- [Vision](#)
- [PDF support](#)
- [Files API](#)
- [Search results](#)
- [Google Sheets add-on](#)

Tools

- [Overview](#)
- [How to implement tool use](#)
- [Token-efficient tool use](#)
- [Fine-grained tool streaming](#)
- [Bash tool](#)
- [Code execution tool](#)
- [Computer use tool](#)
- [Text editor tool](#)
- [Web search tool](#)

Model Context Protocol (MCP)

- [MCP connector](#)

- [Remote MCP servers](#)

Use Cases

- [Overview](#)
- [Ticket routing](#)
- [Customer support agent](#)
- [Content moderation](#)
- [Legal summarization](#)

Prompt Engineering

- [Overview](#)
- [Claude 4 best practices](#)
- [Prompt generator](#)
- [Use prompt templates](#)
- [Prompt improver](#)
- [Be clear and direct](#)
- [Use examples \(multishot prompting\)](#)
- [Let Claude think \(CoT\)](#)
- [Use XML tags](#)
- [Give Claude a role \(system prompts\)](#)
- [Prefill Claude's response](#)
- [Chain complex prompts](#)
- [Long context tips](#)
- [Extended thinking tips](#)

Test & Evaluate

- [Define success criteria](#)
- [Develop test cases](#)
- [Using the Evaluation Tool](#)
- [Reducing latency](#)

Strengthen Guardrails

- [Reduce hallucinations](#)
- [Increase output consistency](#)
- [Mitigate jailbreaks](#)
- [Streaming refusals](#)
- [Reduce prompt leak](#)
- [Keep Claude in character](#)

Legal Center

- [Anthropic Privacy Policy](#)
- [Security and compliance](#)

Intro to Claude - Anthropic

First steps

Intro to Claude

Claude is a highly performant, trustworthy, and intelligent AI platform built by Anthropic. Claude excels at tasks involving language, reasoning, analysis, coding, and more.

Introducing Claude 4, our latest generation of models:

Claude Opus 4 - Our most capable and intelligent model yet. Claude Opus 4 sets new standards in complex reasoning and advanced coding.

Claude Sonnet 4 - Our high-performance model with exceptional reasoning and efficiency.

Learn more in our [blog post](#).

Looking to chat with Claude? Visit [claude.ai](#)!

Get started

If you're new to Claude, start here to learn the essentials and make your first API call.

- [Set up your development environment for building with Claude.](#)
 - [Learn about the family of Claude models.](#)
 - [Explore example prompts for inspiration.](#)
-

Develop with Claude

Anthropic has best-in-class developer tools to build scalable applications with Claude.

- [Enjoy easier, more powerful prompting in your browser with the Workbench and prompt generator tool.](#)
 - [Explore, implement, and scale with the Anthropic API and SDKs.](#)
 - [Learn with interactive Jupyter notebooks that demonstrate uploading PDFs, embeddings, and more.](#)
-

Key capabilities

Claude can assist with many tasks that involve text, code, and images.

- [Text and code generation](#): Summarize text, answer questions, extract data, translate text, and explain and generate code.
 - [Vision](#): Process and analyze visual input and generate text and code from images.
-

Support

- [Help Center](#): Find answers to frequently asked account and billing questions.
- [Service Status](#): Check the status of Anthropic services.

Was this page helpful?

Yes No

[Get started](#)

Follow and Connect

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [Get started](#)
- [Develop with Claude](#)
- [Key capabilities](#)
- [Support](#)

Get started with Claude - Anthropic

First steps

Get started with Claude

Let's learn how to use the Anthropic API to build with Claude.

In this example, we'll explore how to use Claude through the Console and API, starting with a simple question and then customizing Claude's responses.

Prerequisites

You will need:

- An Anthropic [Console account](#)
- An [API key](#)
- Python 3.7+ or TypeScript 4.5+

Anthropic provides [Python and TypeScript SDKs](#), although you can make direct HTTP requests to the API.

Start with the Workbench

Any API call you make—regardless of the specific task—sends a well-configured prompt to the Anthropic API. As you're learning to make the most of Claude, we recommend that you start the development process in the Workbench, a web-based interface to Claude.

Log into the [Anthropic Console](#) and click **Write a prompt from scratch**.

In the middle section, under User, let's ask Claude a question.

[Try in Console](#)

Why is the ocean salty?

Click **Run**. On the right side, you'll see output like

Response:

The ocean is salty due to several factors:

1. **Weathering of rocks:** Over millions of years, rain, rivers, and streams have eroded
2. **Volcanic activity:** Underwater volcanoes and hydrothermal vents release minerals, in
3. **Atmospheric deposition:** Salt particles from ocean spray can be carried by wind and
4. **Evaporation:** As water evaporates from the surface of the ocean, it leaves behind di
5. **Biological processes:** Some marine organisms contribute to the ocean's salinity by r

Over time, these processes have continuously added salts to the ocean, while evaporati

This is a good answer, but let's say we wanted to control the exact type of answer Claude gives. For example, only allowing Claude to respond to questions with poems. We can control the format, tone, and personality of the response by adding a System Prompt.

[Try in Console](#)

You are a world-class poet. Respond only with short poems.

Click **Run** again.

Response:

```
The ocean's salty brine,  
A tale of time and elements combined.  
Rocks and rain, a slow erosion,  
Minerals carried in solution.  
Eons pass, the salt remains,  
In the vast, eternal watery domain.
```

See how Claude's response has changed? LLMs respond well to clear and direct instructions. You can put the role instructions in either the system prompt or the user message. We recommend testing to see which way yields the best results for your use case.

Once you've tweaked the inputs such that you're pleased with the output and have a good sense how to use Claude, convert your Workbench into an integration.

[Click Get Code](#) to copy the generated code representing your Workbench session.

Install the SDK

Anthropic provides SDKs for [Python](#) (3.7+), [TypeScript](#) (4.5+), and [Java](#) (8+). We also currently have a [Go](#) SDK in beta.

- Python
- TypeScript
- Java

In your project directory, create a virtual environment.

```
python -m venv claude-env
```

Activate the virtual environment using:

- On macOS or Linux: `source claude-env/bin/activate`
- On Windows: `claude-env\Scripts\activate`

Install the SDK:

```
pip install anthropic
```

For TypeScript:

```
npm install @anthropic-ai/sdk
```

For Java, find the current version on [Maven Central](#), then add as dependency:

Gradle:

```
implementation 'com.anthropic:anthropic-java:1.0.0'
```

Maven:

```
<dependency>  
  <groupId>com.anthropic</groupId>  
  <artifactId>anthropic-java</artifactId>  
  <version>1.0.0</version>  
</dependency>
```

Set your API key

Every API call requires a valid API key. The SDKs are designed to pull the API key from an environmental variable `ANTHROPIC_API_KEY`. You can also supply the key when initializing the client.

macOS and Linux:

```
export ANTHROPIC_API_KEY='your-api-key-here'
```

Windows:

```
set ANTHROPIC_API_KEY=your-api-key-here
```

Call the API

Call the API by passing the proper parameters to the [messages](#) endpoint.

Note that the code provided by the Workbench sets the API key in the constructor. If you set the API key as an environment variable, you can omit that line as below.

Python

```
import anthropic

client = anthropic.Anthropic()

message = client.messages.create(
    model="claude-opus-4-20250514",
    max_tokens=1000,
    temperature=1,
    system="You are a world-class poet. Respond only with short poems.",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": "Why is the ocean salty?"
                }
            ]
        }
    ]
)

print(message.content)
```

TypeScript

```
import anthropic from 'anthropic';

const client = anthropic;

(async () => {
    const message = await client.messages.create({
        model: "claude-opus-4-20250514",
```



```

    max_tokens: 1000,
    temperature: 1,
    system: "You are a world-class poet. Respond only with short poems.",
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: "Why is the ocean salty?" }],
      },
    ],
  });

  console.log(message.content);
}());

```

Java

```

// (Assuming the Anthropic SDK is properly imported and configured)
import com.anthropic.Client;
import com.anthropic.Message;

public class Quickstart {
  public static void main(String[] args) {
    Client client = new Client();

    Message message = client.messages.create(
      "claude-opus-4-20250514",
      1000,
      1,
      "You are a world-class poet. Respond only with short poems.",
      new Message[] {
        new Message("user", "Why is the ocean salty?")
      }
    );

    System.out.println(message.getContent());
  }
}

```

Run the code using:

- `python3 claude_quickstart.py`
- `node claude_quickstart.js`
- Compile and run your Java code accordingly.

The Workbench and code examples use default model settings for: model (name), temperature, and max tokens to sample.

This quickstart shows how to develop a basic, but functional, Claude-powered application using the Console, Workbench, and API. You can use this same workflow as the foundation for much more powerful use cases.

Next steps

Now that you have made your first Anthropic API request, it's time to explore what else is possible:

- [Use Case Guides](#): End to end implementation guides for common use cases.

- [Anthropic Cookbook](#): Learn with interactive Jupyter notebooks that demonstrate uploading PDFs, embeddings, and more.
 - [Prompt Library](#): Explore dozens of example prompts for inspiration across use cases.
-

Was this page helpful?

[Yes](#) | [No](#)

[Intro to Claude](#) | [Models overview](#)

Follow us on:

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [Prerequisites](#)
- [Start with the Workbench](#)
- [Install the SDK](#)
- [Set your API key](#)
- [Call the API](#)
- [Next steps](#)

Models overview - Anthropic

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello



- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello



- Hello
- World

1. Hello
2. World

Hello World

Hello World

Hello World

Hello World

Models overview - Anthropic

Introduction

Claude is a family of state-of-the-art large language models developed by Anthropic. This guide introduces our models and compares their performance with legacy models.

Latest Models

- **Claude Opus 4** - Our most capable and intelligent model yet. Claude Opus 4 sets new standards in complex reasoning and advanced coding.
- **Claude Sonnet 4** - Our high-performance model with exceptional reasoning and efficiency.

Learn more in our [blog post](#).

Model Comparison

Claude Opus 4

- Our most powerful and capable model
- Inputs: Text and image
- Outputs: Text
- Context window: 200k tokens
- Superior reasoning capabilities

Claude Sonnet 4

- High-performance model with exceptional reasoning
- Inputs: Text and image
- Outputs: Text
- Context window: 200k tokens

Model Names

Model	Anthropic API	AWS Bedrock	GCP Vertex AI
Claude Opus 4	claude-opus-4-20250514	anthropic.claude-opus-4-20250514-v1:0	claude-opus-4@20250514
Claude Sonnet 4	claude-sonnet-4-20250514	anthropic.claude-sonnet-4-20250514-v1:0	claude-sonnet-4@20250514
Claude Sonnet 3.7	claude-3-7-sonnet-20250219 (claude-3-7-sonnet-latest)	anthropic.claude-3-7-sonnet-20250219-v1:0	claude-3-7-sonnet@20250219
Claude Haiku 3.5	claude-3-5-haiku-20241022 (claude-3-5-haiku-latest)	anthropic.claude-3-5-haiku-20241022-v1:0	claude-3-5-haiku@20241022

Models with the same snapshot date are identical across all platforms and do not change. The snapshot date ensures consistency.

Model Aliases

Model	Alias	Model ID
Claude Opus 4	claude-opus-4-0	claude-opus-4-20250514
Claude Sonnet 4	claude-sonnet-4-0	claude-sonnet-4-20250514
Claude Sonnet 3.7	claude-3-7-sonnet-latest	claude-3-7-sonnet-20250219
Claude Sonnet 3.5	claude-3-5-sonnet-latest	claude-3-5-sonnet-20241022
Claude Haiku 3.5	claude-3-5-haiku-latest	claude-3-5-haiku-20241022

Aliases point to the latest snapshots and are subject to same rate limits and pricing.

Model Comparison Table

Feature	Claude Opus 4	Claude Sonnet 4	Claude Sonnet 3.7	Claude Sonnet 3.5	Claude Haiku 3.5	Claude Opus 3	Claude Haiku 3
Description	Most capable	High-performance	Early extended thinking	Previous model	Fastest	Complex tasks	Quick & compact
Strengths	Highest intelligence	Balanced performance	Extended thinking toggle	Capability	Speed	Fluency and understanding	Instant responsiveness
Multilingual	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Vision	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Extended thinking	Yes	Yes	Yes	No	No	No	No
Priority Tier	Yes	Yes	Yes	Yes	Yes	No	No
API model name	claude-opus-4-20250514	claude-sonnet-4-20250514	claude-sonnet-3-7-sonnet-20250219	Upgraded: claude-sonnet-3-5-20241022 Previous: claude-sonnet-3-5-sonnet-20240620	claude-sonnet-3-5-20241022	claude-3-opus-20240229	claude-3-haiku-20240307
Latency	Moderately fast	Fast	Fast	Fast	Fastest	Moderately fast	Fast
Context window	200K	200K	200K	200K	200K	200K	200K
Max output	32,000 tokens	64,000 tokens	64,000 tokens	8,192 tokens	8,192 tokens	4,096 tokens	4,096 tokens
Training data cutoff	Mar 2025	Mar 2025	Nov 2024	Apr 2024	July 2024	Aug 2023	Aug 2023

Note: The knowledge cutoff varies slightly, especially for models trained on recent data.

Migration to Claude 4

Switching involves minimal changes:

- Update your model name:
 - From: claude-3-7-sonnet-20250219
 - To: claude-sonnet-4-20250514 or claude-opus-4-20250514
- Existing API calls continue to work, but behavior has changed slightly. See [API release notes](#).

Learn more at [Migrating to Claude 4](#).

Get started with Claude

Explore Claude's capabilities and development flow, whether you're developing apps or testing AI interactively.

- Visit [claude.ai](#) to chat with Claude.
- [Intro to Claude](#): Understand how Claude works.
- [Quickstart](#): Make your first API call quickly.
- Use the [Anthropic Console](#) to craft and test prompts.

Questions? Contact our [support team](#) or join our [Discord community](#).

On this page

- [Model names](#)
- [Model aliases](#)
- [Model comparison table](#)
- [Model pricing](#)
- [Prompt and output performance](#)
- [Migrating to Claude 4](#)
- [Get started with Claude](#)

Choosing the right model - Anthropic

Introduction

Selecting the optimal Claude model for your application involves balancing three key considerations: capabilities, speed, and cost. This guide helps you make an informed decision based on your specific requirements.

Establish key criteria

When choosing a Claude model, we recommend first evaluating these factors:

- **Capabilities:** What specific features or capabilities will you need the model to have in order to meet your needs?
- **Speed:** How quickly does the model need to respond in your application?
- **Cost:** What's your budget for both development and production usage?

Knowing these answers in advance will make narrowing down and deciding which model to use much easier.

Choose the best model to start with

There are two general approaches you can use to start testing which Claude model best works for your needs.

Option 1: Start with a fast, cost-effective model

For many applications, starting with a faster, more cost-effective model like Claude 3.5 Haiku can be the optimal approach:

1. Begin implementation with Claude 3.5 Haiku
2. Test your use case thoroughly
3. Evaluate if performance meets your requirements
4. Upgrade only if necessary for specific capability gaps

This approach allows for quick iteration, lower development costs, and is often sufficient for many common applications. This approach is best for:

- Initial prototyping and development
- Applications with tight latency requirements
- Cost-sensitive implementations
- High-volume, straightforward tasks

Option 2: Start with the most capable model

For complex tasks where intelligence and advanced capabilities are paramount, you may want to start with the most capable model and then consider optimizing to more efficient models down the line:

1. Implement with Claude Opus 4 or Claude Sonnet 4
2. Optimize your prompts for these models
3. Evaluate if performance meets your requirements

4. Consider increasing efficiency by downgrading intelligence over time with greater workflow optimization

This approach is best for:

- Complex reasoning tasks
- Scientific or mathematical applications
- Tasks requiring nuanced understanding
- Applications where accuracy outweighs cost considerations
- Advanced coding

Model selection matrix

When you need...	We recommend starting with...	Example use cases
Highest intelligence and reasoning, superior capabilities for the most complex tasks, such as multi agent coding	Claude Opus 4	Multi agent frameworks, complex codebase refactoring, nuanced creative writing, complex financial or scientific analysis
Balance of intelligence and speed, strong performance but with faster response times	Claude Sonnet 4	Complex customer chatbot inquiries, complex code generation, straightforward agentic loops, data analysis
Fast responses at lower cost, optimized for high volume, straightforward applications with no need for extended thinking	Claude 3.5 Haiku	Basic customer support, high volume formulaic content generation, straightforward data extraction

Decide whether to upgrade or change models

To determine if you need to upgrade or change models, you should:

1. [Create benchmark tests](#) specific to your use case - having a good evaluation set is the most important step in the process
 2. Test with your actual prompts and data
 3. Compare performance across models for:
 - *Accuracy of responses*
 - *Response quality*
 - *Handling of edge cases*
 4. Weigh performance and cost tradeoffs
-

Next steps

- [Model comparison chart](#)
See detailed specifications and pricing for the latest Claude models
 - [Migrate to Claude 4](#)
Follow the checklist for an easy migration to Claude 4
 - [Start building](#)
Get started with your first API call
-

Feedback

Was this page helpful?

Yes No

Related links

- [Models overview](#)
- [Migrating to Claude 4](#)

Connect with us

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [Establish key criteria](#)
- [Choose the best model to start with](#)
- [Option 1: Start with a fast, cost-effective model](#)
- [Option 2: Start with the most capable model](#)
- [Model selection matrix](#)
- [Decide whether to upgrade or change models](#)
- [Next steps](#)

Migrating to Claude 4 - Anthropic

Overview

This page provides guidance on migrating from Claude 3.7 models to Claude 4 models (Opus 4 and Sonnet 4).

In most cases, you can switch to Claude 4 models with minimal changes:

1. Update your model name:

- From: `claude-3-7-sonnet-20250219`
- To: `claude-sonnet-4-20250514` or `claude-opus-4-20250514`

2. Existing API calls should continue to work without modification, although API behavior has changed slightly in Claude 4 models (see [API release notes](#) for details).

What's new in Claude 4

New refusal stop reason

Claude 4 models introduce a new `refusal` stop reason for content that the model declines to generate for safety reasons, due to the increased intelligence of Claude 4 models:

```
{
  "id": "msg_014XEDjypDjFzgKVWdFUXxZP",
  "type": "message",
  "role": "assistant",
  "model": "claude-sonnet-4-20250514",
  "content": [{"type": "text", "text": "I would be happy to assist you. You can "}],
  "stop_reason": "refusal",
  "stop_sequence": null,
  "usage": {
    "input_tokens": 564,
    "cache_creation_input_tokens": 0,
    "cache_read_input_tokens": 0,
    "output_tokens": 22
  }
}
```

When migrating to Claude 4, you should update your application to [handle](#) `refusal` stop reasons.

Summarized thinking

With extended thinking enabled, the Messages API for Claude 4 models returns a summary of Claude's full thinking process. Summarized thinking provides the full intelligence benefits of extended thinking, while preventing misuse.

While the API is consistent across Claude 3.7 and 4 models, streaming responses for extended thinking might return in a "chunky" delivery pattern, with possible delays between streaming events.

Summarization is processed by a different model than the one you target in your requests. The thinking model does not see the summarized output.

For more information, see the [Extended thinking documentation](#).

Interleaved thinking

Claude 4 models support interleaving tool use with extended thinking, allowing for more natural conversations where tool uses and responses can be mixed with regular messages.

Interleaved thinking is in beta. To enable it, add the [beta header](#) `interleaved-thinking-2025-05-14` to your API request.

For more information, see the [Extended thinking documentation](#).

Updated text editor tool

The text editor tool has been updated for Claude 4 models with the following changes:

- **Tool type:** `text_editor_20250429`
- **Tool name:** `str_replace_based_edit_tool`
- The `undo_edit` command is no longer supported in Claude 4 models.

The `str_replace_editor` text editor tool remains the same for Claude Sonnet 3.7.

If migrating from Claude Sonnet 3.7 and using the text editor tool:

```
# Claude Sonnet 3.7
tools=[
  {
    "type": "text_editor_20250124",
    "name": "str_replace_editor"
  }
]

# Claude 4
tools=[
  {
    "type": "text_editor_20250429",
    "name": "str_replace_based_edit_tool"
  }
]
```

For more information, see the [Text editor tool documentation](#).

Token-efficient tool use no longer supported

[Token-efficient tool use](#) is only available in Claude Sonnet 3.7.

If migrating from Claude Sonnet 3.7 and using token-efficient tool use, we recommend removing the `token-efficient-tools-2025-02-19` [beta header](#) from your requests.

The `token-efficient-tools-2025-02-19` beta header can still be included in Claude 4 requests, but it will have no effect.

Extended output no longer supported

The `output-128k-2025-02-19` [beta header](#) for extended output is only available in Claude Sonnet 3.7.

If migrating from Claude Sonnet 3.7, we recommend removing `output-128k-2025-02-19` from your requests.

The `output-128k-2025-02-19` beta header can still be included in Claude 4 requests, but it will have no effect.

Performance considerations

Claude Sonnet 4

- Improved reasoning and intelligence capabilities compared to Claude Sonnet 3.7
- Enhanced tool use accuracy

Claude Opus 4

- Most capable model with superior reasoning and intelligence
- Slower than Sonnet models
- Best for complex tasks requiring deep analysis

Migration checklist

- Update model id in your API calls
- Test existing requests (should work without changes)
- Remove `token-efficient-tools-2025-02-19` beta header if applicable
- Remove `output-128k-2025-02-19` beta header if applicable
- Handle new `refusal` stop reason
- Update text editor tool type and name if using it
- Remove any code that uses the `undo_edit` command
- Explore new tool interleaving capabilities with extended thinking
- Review [Claude 4 prompt engineering best practices](#)
- Test in development before production deployment

Need help?

- Check our [API documentation](#) for detailed specifications.
- Review [model capabilities](#) for performance comparisons.
- Review [API release notes](#) for updates.
- Contact support if you encounter issues during migration.

Was this page helpful?

- [Yes](#)
- [No](#)

[Choosing a model](#)

[Model deprecations](#)

Follow us: [!\[\]\(a8f9309f944226d1420f5fed22e2b6e6_img.jpg\) !\[\]\(729993f67f08b74df2e570c12c53c92e_img.jpg\) !\[\]\(1c2527607ca033f0a9f29d0e6fa5c358_img.jpg\)](#)

Model deprecations - Anthropic

Overview

Anthropic uses the following terms to describe the lifecycle of our models:

- **Active:** The model is fully supported and recommended for use.
- **Legacy:** The model will no longer receive updates and may be deprecated in the future.
- **Deprecated:** The model is no longer available for new customers but continues to be available for existing users until retirement. We assign a retirement date at this point.
- **Retired:** The model is no longer available for use. Requests to retired models will fail.

Migrating to replacements

Once a model is deprecated, please migrate all usage to a suitable replacement before the retirement date. Requests to models past the retirement date will fail.

To help measure the performance of replacement models on your tasks, we recommend thorough testing of your applications with the new models well before the retirement date.

For specific instructions on migrating from Claude 3.7 to Claude 4 models, see [Migrating to Claude 4](#).

Notifications

Anthropic notifies customers with active deployments for models with upcoming retirements. We provide at least 6 months[†] notice before model retirement for publicly released models.

Auditing model usage

To help identify usage of deprecated models, customers can access an audit of their API usage. Follow these steps:

1. Go to <https://console.anthropic.com/settings/usage>
2. Click the “Export” button
3. Review the downloaded CSV to see usage broken down by API key and model

This audit will help you locate any instances where your application is still using deprecated models, allowing you to prioritize updates to newer models before the retirement date.

Model status

All publicly released models are listed below with their status:

API Model Name	Current State	Deprecated	Retired
claude-1.0	Retired	September 4, 2024	November 6, 2024
claude-1.1	Retired	September 4, 2024	November 6, 2024
claude-1.2	Retired	September 4, 2024	November 6, 2024
claude-1.3	Retired	September 4, 2024	November 6, 2024
claude-instant-1.0	Retired	September 4, 2024	November 6, 2024
claude-instant-1.1	Retired	September 4, 2024	November 6, 2024

API Model Name	Current State	Deprecated	Retired
claude-instant-1.2	Retired	September 4, 2024	November 6, 2024
claude-2.0	Deprecated	January 21, 2025	Expected Jul 2025
claude-2.1	Deprecated	January 21, 2025	Expected Jul 2025
claude-3-sonnet-20240229	Deprecated	January 21, 2025	Expected Jul 2025
claude-3-opus-20240229	Deprecated	June 30, 2025	Expected Jan 2026
claude-3-haiku-20240307	Active	N/A	N/A
claude-3-5-sonnet-20240620	Active	N/A	N/A
claude-3-5-haiku-20241022	Active	N/A	N/A
claude-3-5-sonnet-20241022	Active	N/A	N/A
claude-3-7-sonnet-20250219	Active	N/A	N/A
claude-sonnet-4-20250514	Active	N/A	N/A
claude-opus-4-20250514	Active	N/A	N/A

Deprecation history

All deprecations are listed below, with the most recent announcements at the top.

2025-06-30: Claude Opus 3 model

On June 30, 2025, we notified developers using Claude Opus 3 model of its upcoming retirement.

Retirement Date	Deprecated Model	Recommended Replacement
January 2026	claude-3-opus-20240229	claude-opus-4-20250514

2025-01-21: Claude 2, Claude 2.1, and Claude Sonnet 3 models

On January 21, 2025, we notified developers using Claude 2, Claude 2.1, and Claude Sonnet 3 models of their upcoming retirements.

Retirement Date	Deprecated Model	Recommended Replacement
July 21, 2025	claude-2.0	claude-3-5-sonnet-20241022
July 21, 2025	claude-2.1	claude-3-5-sonnet-20241022
July 21, 2025	claude-3-sonnet-20240229	claude-3-5-sonnet-20241022

2024-09-04: Claude 1 and Instant models

On September 4, 2024, we notified developers using Claude 1 and Instant models of their upcoming retirements.

Retirement Date	Deprecated Model	Recommended Replacement
November 6, 2024	claude-1.0	claude-3-5-haiku-20241022
November 6, 2024	claude-1.1	claude-3-5-haiku-20241022
November 6, 2024	claude-1.2	claude-3-5-haiku-20241022
November 6, 2024	claude-1.3	claude-3-5-haiku-20241022
November 6, 2024	claude-instant-1.0	claude-3-5-haiku-20241022
November 6, 2024	claude-instant-1.1	claude-3-5-haiku-20241022
November 6, 2024	claude-instant-1.2	claude-3-5-haiku-20241022

Best practices

1. Regularly check our documentation for updates on model deprecations.
2. Test your applications with newer models well before the retirement date of your current model.
3. Update your code to use the recommended replacement model as soon as possible.
4. Contact our support team if you need assistance with migration or have any questions.

† The Claude 1 family of models have a 60-day notice period due to their limited usage compared to our newer models.

On this page

- [Overview](#)
- [Migrating to replacements](#)
- [Notifications](#)
- [Auditing model usage](#)
- [Model status](#)
- [Deprecation history](#)
- [2025-06-30: Claude Opus 3 model](#)
- [2025-01-21: Claude 2, Claude 2.1, and Claude Sonnet 3 models](#)
- [2024-09-04: Claude 1 and Instant models](#)
- [Best practices](#)

Pricing

Learn about Anthropic's pricing structure for models and features

This page provides detailed pricing information for Anthropic's models and features. All prices are in USD.

For the most current pricing information, please visit anthropic.com/pricing.

Model pricing

The following table shows pricing for all Claude models across different usage tiers:

Model	Base Input Tokens	5m Cache Writes	1h Cache Writes	Cache Hits & Refreshes	Output Tokens
Claude Opus 4	\$15 / MTok	\$18.75 / MTok	\$30 / MTok	\$1.50 / MTok	\$75 / MTok
Claude Sonnet 4	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Sonnet 3.7	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Sonnet 3.5	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Haiku 3.5	\$0.80 / MTok	\$1 / MTok	\$1.6 / MTok	\$0.08 / MTok	\$4 / MTok
Claude Opus 3	\$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

MTok = Million tokens. The "Base Input Tokens" column shows standard input pricing, "Cache Writes" and "Cache Hits" are specific to [prompt caching](#), and "Output Tokens" shows output pricing.

Feature-specific pricing

Batch processing

The Batch API allows asynchronous processing of large volumes of requests with a 50% discount on both input and output tokens.

Model	Batch input	Batch output
Claude Opus 4	\$7.50 / MTok	\$37.50 / MTok
Claude Sonnet 4	\$1.50 / MTok	\$7.50 / MTok
Claude Sonnet 3.7	\$1.50 / MTok	\$7.50 / MTok
Claude Sonnet 3.5	\$1.50 / MTok	\$7.50 / MTok
Claude Haiku 3.5	\$0.40 / MTok	\$2 / MTok
Claude Opus 3	\$7.50 / MTok	\$37.50 / MTok
Claude Haiku 3	\$0.125 / MTok	\$0.625 / MTok

For more information, see [batch processing documentation](#).

Tool use pricing

Tool use requests are priced based on:

1. The total number of input tokens sent to the model (including in the `tools` parameter)
2. The number of output tokens generated
3. For server-side tools, additional usage-based pricing (e.g., web search charges per search performed)

Client-side tools are priced the same as other Claude API requests, while server-side tools may incur additional charges based on their usage.

The additional tokens from tool use originate from:

- The `tools` parameter in API requests (tool names, descriptions, schemas)
- `tool_use` content blocks in API requests and responses
- `tool_result` content blocks in API requests

When using `tools`, a special system prompt is automatically included for the model, enabling tool use. The number of tool use tokens required per model is shown below (excluding extra tokens listed above). If no `tools` are provided, the `none` option uses 0 additional system prompt tokens.

Model	Tool choice	Tool use system prompt token count
Claude Opus 4	auto, none, any, tool	346 tokens / 313 tokens
Claude Sonnet 4	auto, none, any, tool	346 tokens / 313 tokens
Claude Sonnet 3.7	auto, none, any, tool	346 tokens / 313 tokens
Claude Sonnet 3.5 (Oct)	auto, none, any, tool	346 tokens / 313 tokens
Claude Sonnet 3.5 (June)	auto, none, any, tool	294 tokens / 261 tokens
Claude Haiku 3.5	auto, none, any, tool	264 tokens / 340 tokens
Claude Opus 3	auto, none, any, tool	530 tokens / 281 tokens
Claude Sonnet 3	auto, none, any, tool	159 tokens / 235 tokens
Claude Haiku 3	auto, none, any, tool	264 tokens / 340 tokens

These token counts are added to your normal input and output tokens to calculate total request cost.

For current model prices, see [model pricing](#).

For more on tool use implementation and best practices, see [tool use documentation](#).

Specific tool pricing

Bash tool

The bash tool adds **245 input tokens** to your API calls.

Additional tokens are consumed by:

- Command outputs (stdout/stderr)
- Error messages
- Large file contents

See [tool use pricing](#) for complete details.

Code execution tool

Usage is tracked separately, with execution time of at least 5 minutes.

If files are included, billing applies even if the tool is not used due to preloaded files.

Pricing: \$0.05 per session-hour.

Text editor tool

Uses the same pricing as other tools with:

- Standard input/output token pricing
- Additional input tokens depending on model:

Tool	Additional input tokens
text_editor_20250429 (Claude 4)	700 tokens
text_editor_20250124 (Claude Sonnet 3.7)	700 tokens
text_editor_20241022 (Claude Sonnet 3.5)	700 tokens

See [tool use pricing](#) for details.

Web search tool

Charges in addition to token costs:

```
{
  "usage": {
    "input_tokens": 105,
    "output_tokens": 6039,
    "cache_read_input_tokens": 7123,
    "cache_creation_input_tokens": 7345,
    "server_tool_use": {
      "web_search_requests": 1
    }
  }
}
```

Web search costs **\$10/1000 searches**, plus standard token costs. Results count as input tokens, including in ongoing conversation turns. Each search counts as one use; failed searches are free.

Computer use tool

Follows [standard tool use pricing](#).

- **System prompt overhead:** 466-499 tokens.
- **Input tokens per model:**

Model	Input tokens per definition
Claude 4 / Sonnet 3.7	735 tokens
Claude Sonnet 3.5	683 tokens

- Additional token costs include screenshots (see [Vision pricing](#)) and tool execution results.

If using bash or text editor tools simultaneously, their costs apply as well.

Agent use case pricing examples

Customer support agent example

Estimating cost for 10,000 tickets:

- Approximately 3,700 tokens per conversation
- Using Claude Sonnet 4 at around 3/MTok input and 15/MTok output

Total estimate: about \$22.20 for 10,000 tickets.

See [customer support agent guide](#) for details.

General agent workflow pricing

For complex multi-step workflows:

1. **Initial request:** 500-1,000 tokens, ~\$0.003 per request.
2. **Memory retrieval:** 2,000-5,000 tokens, ~\$0.015 per retrieval.
3. **Action planning/execution:**
 - Planning: 1,000-2,000 tokens.
 - Feedback: 500-1,000 tokens.
 - Total: ~\$0.045 per action.

See [agent use cases guide](#) for details.

Cost optimization strategies

- Use the appropriate models: Haiku for simple tasks, Sonnet for complex reasoning.
- Implement prompt caching.
- Batch requests via the Batch API.
- Monitor token usage patterns.

For large-scale applications, contact [our sales team](#).

Additional pricing considerations

Rate limits

Vary by tier:

- Tier 1-4, and custom enterprise limits.

See [rate limits documentation](#).

Volume discounts

Negotiated on case-by-case basis. Contact sales via [email](#).

Enterprise pricing

Custom arrangements, including:

- Rate limits
- Volume discounts
- Dedicated support

Contact [sales](#) or through [Console](#).

Billing and payment

- Monthly billing based on actual usage.
- Payments in USD.
- Credit card and invoicing options.
- Usage tracked in [Console](#).

Frequently asked questions

How is token usage calculated?

Tokens are pieces of text processed by models: roughly 4 characters or 0.75 words in English.

Are there free tiers or trials?

Yes, new users get free credits; contact sales for extended trials.

How do discounts stack?

Examples include combined use of batch API and prompt caching.

What payment methods are accepted?

Major credit cards are accepted; enterprise invoicing is available.

For more info, contact [support](#).

Just a moment...

Enable JavaScript and cookies to continue

Trust Center

Building with Claude - Anthropic

Learn about Claude

Building with Claude

Claude is a family of [highly performant and intelligent AI models](#) built by Anthropic. While Claude is powerful and extensible, it's also the most trustworthy and reliable AI available. It follows critical protocols, makes fewer mistakes, and is resistant to jailbreaks—allowing [enterprise customers](#) to build the safest AI-powered applications at scale.

This guide introduces Claude's enterprise capabilities, the end-to-end flow for developing with Claude, and how to start building.

What you can do with Claude

Claude is designed to empower enterprises at scale with strong performance across benchmark evaluations for reasoning, math, coding, and fluency in English and non-English languages.

Here's a non-exhaustive list of Claude's capabilities and common uses.

Capability	Enables you to...
Text and code generation	- Adhere to brand voice for excellent customer-facing experiences such as copywriting and chatbots
	- Create production-level code and operate (in-line code generation, debugging, and conversational querying) within complex codebases
	- Build automatic translation features between languages
	- Conduct complex financial forecasts
Vision	- Support legal use cases that require high-quality technical analysis, long context windows for processing detailed documents, and fast outputs
	- Process and analyze visual input, such as extracting insights from charts and graphs
	- Generate code from images with code snippets or templates based on diagrams
Tool use	- Describe an image for a user with low vision
	- Interact with external client-side tools and functions, allowing Claude to reason, plan, and execute actions by generating structured outputs through API calls

Enterprise considerations

Along with an extensive set of features, tools, and capabilities, Claude is also built to be secure, trustworthy, and scalable for wide-reaching enterprise needs.

Feature	Description
Secure	- Enterprise-grade security and data handling for API
	- SOC II Type 2 certified, HIPAA compliance options for API
	- Accessible through AWS (GA) and GCP (in private preview)
Trustworthy	- Resistant to jailbreaks and misuse. We continuously monitor prompts and outputs for harmful, malicious use cases that violate our AUP .
	- Copyright indemnity protections for paid commercial services
	- Uniquely positioned to serve high trust industries that process large volumes of sensitive user data

Feature	Description
Capable	- 200K token context window for expanded use cases, with future support for 1M
	- Tool use , also known as function calling, which allows seamless integration of Claude into specialized applications and custom workflows
	- Multimodal input capabilities with text output, allowing you to upload images (such as tables, graphs, and photos) along with text prompts for richer context and complex use cases
Reliable	- Developer Console with Workbench and prompt generation tool for easier, more powerful prompting and experimentation
	- SDKs and APIs to expedite and enhance development
Global	- Very low hallucination rates
	- Accurate over long documents
Cost conscious	- Great for coding tasks and fluency in English and non-English languages like Spanish and Japanese
	- Enables use cases like translation services and broader global utility
	- Family of models balances cost, performance, and intelligence

Implementing Claude

1. Scope your use case

- Identify a problem to solve or tasks to automate with Claude.
- Define requirements: features, performance, and cost.

2. Design your integration

- Select Claude's capabilities (e.g., vision, tool use) and models (Opus, Sonnet, Haiku) based on needs.
- Choose a deployment method, such as the [Anthropic API](#), AWS Bedrock, or Vertex AI.

3. Prepare your data

- Identify and clean relevant data (databases, code repos, knowledge bases) for Claude's context.

4. Develop your prompts

- Use [Workbench](#) to create evals, draft prompts, and iteratively refine based on test results.
- Deploy polished prompts and monitor real-world performance for further refinement.

5. Implement Claude

- Set up your environment, integrate Claude with your systems (APIs, databases, UIs), and define human-in-the-loop requirements.

6. Test your system

- Conduct red teaming for potential misuse and A/B test improvements.

7. Deploy to production

- Once your application runs smoothly end-to-end, deploy to production.

8. Monitor and improve

- Monitor performance and effectiveness to make ongoing improvements.

Start building with Claude

When you're ready, start building with Claude:

- Follow the [Quickstart](#) to make your first API call
 - Check out the [API Reference](#)
 - Explore the [Prompt Library](#) for example prompts
 - Experiment and start building with the [Workbench](#)
 - Check out the [Anthropic Cookbook](#) for working code examples
-

Was this page helpful?

[Yes](#) [No](#)

[Pricing](#)

[Features overview](#)

On this page

- [What you can do with Claude](#)
 - [Enterprise considerations](#)
 - [Implementing Claude](#)
 - [Start building with Claude](#)
-

Follow us on [X](#), [LinkedIn](#), and [Discord](#).

Features overview - Anthropic

Learn about Claude

Features overview

Explore Claude's advanced features and capabilities.

Core capabilities

These features enhance Claude's fundamental abilities for processing, analyzing, and generating content across various formats and use cases.

Feature	Description	Availability
Batch processing	Process large volumes of requests asynchronously for cost savings. Send batches with a large number of queries per batch. Batch API calls costs 50% less than standard API calls.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Citations	Ground Claude's responses in source documents. With Citations, Claude can provide detailed references to the exact sentences and passages it uses to generate responses, leading to more verifiable, trustworthy outputs.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Extended thinking	Enhanced reasoning capabilities for complex tasks, providing transparency into Claude's step-by-step thought process before delivering its final answer.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Files API	Upload and manage files to use with Claude without re-uploading content with each request. Supports PDFs, images, and text files.	Anthropic API (Beta)
PDF support	Process and analyze text and visual content from PDF documents.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Prompt caching	Provide Claude with more background knowledge and example outputs to reduce costs and latency.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Search results	Enable natural citations for RAG applications by providing search results with proper source attribution. Achieve web search-quality citations for custom knowledge bases and tools.	Anthropic API (Beta)
Token counting	Token counting enables you to determine the number of tokens in a message before sending it to Claude, helping you make informed decisions about your prompts and usage.	Anthropic API Google Cloud's Vertex AI
Tool use	Enable Claude to interact with external tools and APIs to perform a wider variety of tasks. For a list of supported tools, see the Tools table .	Anthropic API Amazon Bedrock Google Cloud's Vertex AI

Tools

These features enable Claude to interact with external systems, execute code, and perform automated tasks through various tool interfaces.

Feature	Description	Availability
Bash	Execute bash commands and scripts to interact with the system shell and perform command-line operations.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Code execution	Run Python code in a sandboxed environment for advanced data analysis.	Anthropic API (Beta)
Computer use	Control computer interfaces by taking screenshots and issuing mouse and keyboard commands.	Anthropic API (Beta) Amazon Bedrock (Beta) Google Cloud's Vertex AI (Beta)
Fine-grained tool streaming	Stream tool use parameters without buffering/JSON validation, reducing latency for receiving large parameters.	Anthropic API Google Cloud's Vertex AI
MCP connector	Connect to remote MCP servers directly from the Messages API without a separate MCP client.	Anthropic API (Beta)
Text editor	Create and edit text files with a built-in text editor interface for file manipulation tasks.	Anthropic API Amazon Bedrock Google Cloud's Vertex AI
Web search	Augment Claude's comprehensive knowledge with current, real-world data from across the web.	Anthropic API

Help & Community

Was this page helpful?

- [Yes](#)
- [No](#)

[Building with Claude](#) | [Context windows](#)

Follow us: [X](#) | [LinkedIn](#) | [Discord](#)

Context windows - Anthropic

Learn about Claude

Context windows

The “context window” refers to the entirety of the amount of text a language model can look back on and reference when generating new text plus the new text it generates. This is different from the large corpus of data the language model was trained on, and instead represents a “working memory” for the model. A larger context window allows the model to understand and respond to more complex and lengthy prompts, while a smaller context window may limit the model’s ability to handle longer prompts or maintain coherence over extended conversations.

The diagram below illustrates the standard context window behavior for API requests^[^1]:

[^1]: For chat interfaces, such as for [claude.ai](#), context windows can also be set up on a rolling “first in, first out” system.

- **Progressive token accumulation:** As the conversation advances through turns, each user message and assistant response accumulates within the context window. Previous turns are preserved completely.
- **Linear growth pattern:** The context usage grows linearly with each turn, with previous turns preserved completely.
- **200K token capacity:** The total available context window (200,000 tokens) represents the maximum capacity for storing conversation history and generating new output from Claude.
- **Input-output flow:** Each turn consists of:
 - **Input phase:** Contains all previous conversation history plus the current user message.
 - **Output phase:** Generates a text response that becomes part of a future input.

The context window with extended thinking

When using [extended thinking](#), all input and output tokens, including the tokens used for thinking, count toward the context window limit, with a few nuances in multi-turn situations.

The thinking budget tokens are a subset of your `max_tokens` parameter, are billed as output tokens, and count towards rate limits.

However, previous thinking blocks are automatically stripped from the context window calculation by the Anthropic API and are not part of the conversation history that the model “sees” for subsequent turns, preserving token capacity for actual conversation content.

The diagram below demonstrates the specialized token management when extended thinking is enabled:

- **Stripping extended thinking:** Extended thinking blocks (shown in dark gray) are generated during each turn’s output phase, **but are not carried forward as input tokens for subsequent turns**. You do not need to strip the thinking blocks yourself. The Anthropic API automatically does this for you if you pass them back.
- **Technical implementation details:**
 - The API automatically excludes thinking blocks from previous turns when you pass them back as part of the conversation history.
 - Extended thinking tokens are billed as output tokens only once, during their generation.
 - The effective context window calculation becomes: $\text{context_window} = (\text{input_tokens} - \text{previous_thinking_tokens}) + \text{current_turn_tokens}$.
 - Thinking tokens include both `thinking` blocks and `redacted_thinking` blocks.

This architecture is token efficient and allows for extensive reasoning without token waste, as thinking blocks can be substantial in length.

You can read more about the context window and extended thinking in our [extended thinking guide](#).

The context window with extended thinking and tool use

The diagram below illustrates the context window token management when combining extended thinking with tool use:

1. First turn architecture

- **Input components:** Tools configuration and user message
- **Output components:** Extended thinking + text response + tool use request
- **Token calculation:** All input and output components count toward the context window, and all output components are billed as output tokens.

2. Tool result handling (turn 2)

- **Input components:** Every block in the first turn as well as the `tool_result`. The extended thinking block **must** be returned with the corresponding tool results. This is the only case wherein you **have to** return thinking blocks.
- **Output components:** After tool results have been passed back to Claude, Claude will respond with only text (no additional extended thinking until the next `user` message).
- **Token calculation:** All input and output components count toward the context window, and all output components are billed as output tokens.

3. Third Step

- **Input components:** All inputs and the output from the previous turn are carried forward, with the exception of the thinking block, which can be dropped since Claude has completed the entire tool use cycle. The API will automatically strip the thinking block if you pass it back, or you can strip it yourself at this stage. This is also where you would add the next `User` turn.
- **Output components:** Since there is a new `User` turn outside of the tool use cycle, Claude will generate a new extended thinking block and continue from there.
- **Token calculation:** Previous thinking tokens are automatically stripped from context window calculations. All other previous blocks still count as part of the token window, and the current turn's thinking block counts as part of the context window.

Considerations for tool use with extended thinking:

- When posting tool results, the entire unmodified thinking block that accompanies that specific tool request (including signature/redacted portions) must be included.
- The effective context window calculation becomes: $\text{context_window} = \text{input_tokens} + \text{current_turn_tokens}$.
- The system uses cryptographic signatures to verify thinking block authenticity. Failing to preserve thinking blocks during tool use can break Claude's reasoning continuity. If you modify thinking blocks, the API will return an error.

Claude 4 models support [interleaved thinking](#), which enables Claude to think between tool calls and make more sophisticated reasoning after receiving tool results.

Claude Sonnet 3.7 does not support interleaved thinking, so there is no interleaving of extended thinking and tool calls without a non-`tool_result` user turn in between.

For more information about using tools with extended thinking, see our [extended thinking guide](#).

Context window management with newer Claude models

In newer Claude models (starting with Claude Sonnet 3.7), if the sum of prompt tokens and output tokens exceeds the model's context window, the system will return a validation error rather than silently truncating the context. This change provides more predictable behavior but requires more careful token management.

To plan your token usage and ensure you stay within context window limits, you can use the [token counting API](#) to estimate how many tokens your messages will use before sending them to Claude.

See our [model comparison](#) table for a list of context window sizes by model.

Next steps

- [Model comparison table](#)
- [Extended thinking overview](#)

Was this page helpful?

Yes | No

- [Features overview](#)
- [Glossary](#)

On this page

- [Understanding the context window](#)
- [The context window with extended thinking](#)
- [The context window with extended thinking and tool use](#)
- [Context window management with newer Claude models](#)
- [Next steps](#)

Learn more about Claude and context windows in the official documentation.

Glossary - Anthropic

Context window

The “context window” refers to the amount of text a language model can look back on and reference when generating new text. This is different from the large corpus of data the language model was trained on, and instead represents a *working memory* for the model. A larger context window allows the model to understand and respond to more complex and lengthy prompts, while a smaller context window may limit the model’s ability to handle longer prompts or maintain coherence over extended conversations.

See our [guide to understanding context windows](#) to learn more.

Fine-tuning

Fine-tuning is the process of further training a pretrained language model using additional data. This causes the model to start representing and mimicking the patterns and characteristics of the fine-tuning dataset. Claude is not a bare language model; it has already been fine-tuned to be a helpful assistant. Our API does not currently offer fine-tuning, but please ask your Anthropic contact if you are interested in exploring this option. Fine-tuning can be useful for adapting a language model to a specific domain, task, or writing style, but it requires careful consideration of the fine-tuning data and the potential impact on the model’s performance and biases.

HHH

These three H’s represent Anthropic’s goals in ensuring that Claude is beneficial to society:

- A **helpful** AI will attempt to perform the task or answer the question posed to the best of its abilities, providing relevant and useful information.
- An **honest** AI will give accurate information, and not hallucinate or confabulate. It will acknowledge its limitations and uncertainties when appropriate.
- A **harmless** AI will not be offensive or discriminatory, and when asked to aid in a dangerous or unethical act, the AI should politely refuse and explain why it cannot comply.

Latency

Latency, in the context of generative AI and large language models, refers to the time it takes for the model to respond to a given prompt. It is the delay between submitting a prompt and receiving the generated output. Lower latency indicates faster response times, which is crucial for real-time applications, chatbots, and interactive experiences. Factors that can affect latency include model size, hardware capabilities, network conditions, and the complexity of the prompt and the generated response.

LLM

Large language models (LLMs) are AI language models with many parameters that are capable of performing a variety of surprisingly useful tasks. These models are trained on vast amounts of text data and can generate human-like text, answer questions, summarize information, and more. Claude is a conversational assistant based on a large language model that has been fine-tuned and trained using RLHF to be more helpful, honest, and harmless.

MCP (Model Context Protocol)

Model Context Protocol (MCP) is an open protocol that standardizes how applications provide context to LLMs. Like a USB-C port for AI applications, MCP provides a unified way to connect AI models to different data sources and tools. MCP enables AI systems to maintain consistent context across interactions and access external resources in a standardized manner. See our [MCP documentation](#) to learn more.

MCP connector

The MCP connector is a feature that allows API users to connect to MCP servers directly from the Messages API without building an MCP client. This enables seamless integration with MCP-compatible tools and services through the Anthropic API. The MCP connector supports features like tool calling and is available in public beta. See our [MCP connector documentation](#) to learn more.

Pretraining

Pretraining is the initial process of training language models on a large unlabeled corpus of text. In Claude's case, autoregressive language models (like Claude's underlying model) are pretrained to predict the next word, given the previous context of text in the document. These pretrained models are not inherently good at answering questions or following instructions, and often require deep skill in prompt engineering to elicit desired behaviors. Fine-tuning and RLHF are used to refine these pretrained models, making them more useful for a wide range of tasks.

RAG (Retrieval augmented generation)

Retrieval augmented generation (RAG) is a technique that combines information retrieval with language model generation to improve the accuracy and relevance of the generated text, and to better ground the model's response in evidence. In RAG, a language model is augmented with an external knowledge base or a set of documents that is passed into the context window. The data is retrieved at run time when a query is sent to the model, although the model itself does not necessarily retrieve the data (but can with [tool use](#) and a retrieval function). When generating text, relevant information first must be retrieved from the knowledge base based on the input prompt, and then passed to the model along with the original query. The model uses this information to guide the output it generates. This allows the model to access and utilize information beyond its training data, reducing reliance on memorization and improving factual accuracy. RAG can be particularly useful for tasks that require up-to-date information, domain-specific knowledge, or explicit citation of sources. However, the effectiveness of RAG depends on the quality and relevance of the external knowledge base and the knowledge that is retrieved at runtime.

RLHF

Reinforcement Learning from Human Feedback (RLHF) is a technique used to train a pretrained language model to behave in ways that are consistent with human preferences. This can include helping the model follow instructions more effectively or act more like a chatbot. Human feedback consists of ranking a set of two or more example texts, and the reinforcement learning process encourages the model to prefer outputs that are similar to the higher-ranked ones. Claude has been trained using RLHF to be a more helpful assistant. For more details, you can read [Anthropic's paper on the subject](#).

Temperature

Temperature is a parameter that controls the randomness of a model's predictions during text generation. Higher temperatures lead to more creative and diverse outputs, allowing for multiple variations in phrasing and, in the case of fiction, variation in answers as well. Lower temperatures result in more conservative and deterministic outputs that stick to the most probable phrasing and answers. Adjusting the temperature enables users to encourage a language model to explore rare, uncommon, or surprising word choices and sequences, rather than only selecting the most likely predictions.

TTFT (Time to first token)

Time to First Token (TTFT) is a performance metric that measures the time it takes for a language model to generate the first token of its output after receiving a prompt. It is an important indicator of the model's responsiveness and is particularly relevant for interactive applications, chatbots, and real-time systems where users expect quick initial feedback. A lower TTFT indicates that the model can start generating a response faster, providing a more seamless and engaging user experience. Factors that can influence TTFT include model size, hardware capabilities, network conditions, and the complexity of the prompt.

Tokens

Tokens are the smallest individual units of a language model, and can correspond to words, subwords, characters, or even bytes (in the case of Unicode). For Claude, a token approximately represents 3.5 English characters, though the exact number can vary depending on the language used. Tokens are typically hidden when interacting with language models at the “text” level but become relevant when examining the exact inputs and outputs of a language model. When Claude is provided with text to evaluate, the text (consisting of a series of characters) is encoded into a series of tokens for the model to process. Larger tokens enable data efficiency during inference and pretraining (and are utilized when possible), while smaller tokens allow a model to handle uncommon or never-before-seen words. The choice of tokenization method can impact the model's performance, vocabulary size, and ability to handle out-of-vocabulary words.

On this page

- [Context window](#)
- [Fine-tuning](#)
- [HHH](#)
- [Latency](#)
- [LLM](#)
- [MCP \(Model Context Protocol\)](#)
- [MCP connector](#)
- [Pretraining](#)
- [RAG \(Retrieval augmented generation\)](#)
- [RLHF](#)
- [Temperature](#)
- [TTFT \(Time to first token\)](#)
- [Tokens](#)

Prompt caching - Anthropic

Capabilities

Prompt caching

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:

```
curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "system": [
      {
        "type": "text",
        "text": "You are an AI assistant tasked with analyzing literary works. Your go
      ]
    "messages": [
      {
        "role": "user",
        "content": "Analyze the major themes in Pride and Prejudice."
      }
    ]
  }'
```

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.

How prompt caching works

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.

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:

Model	Base Input Tokens	5m Cache Writes	1h Cache Writes	Cache Hits & Refreshes	Output Tokens
Claude Opus 4	\$15 / MTok	\$18.75 / MTok	\$30 / MTok	\$1.50 / MTok	\$75 / MTok
Claude Sonnet 4	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Sonnet 3.7	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Sonnet 3.5	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Haiku 3.5	\$0.80 / MTok	\$1 / MTok	\$1.6 / MTok	\$0.08 / MTok	\$4 / MTok
Claude Opus 3	\$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

Note:

- 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
- Regular input and output tokens are priced at standard rates

How to implement prompt caching

Supported models

Prompt caching is currently supported on:

- Claude Opus 4
- Claude Sonnet 4
- Claude Sonnet 3.7
- Claude Sonnet 3.5
- Claude Haiku 3.5
- Claude Haiku 3
- Claude Opus 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 order: `tools`, `system`, then `messages`. This order forms a hierarchy where each level builds on the previous.

Using the `cache_control` parameter, you can define up to 4 cache breakpoints, allowing you to cache different reusable sections separately. For each breakpoint, the system will automatically check for cache hits at previous positions and use the longest matching prefix if one is found.

Cache limitations

The minimum cacheable prompt length is:

- 1024 tokens for Claude Opus 4, Claude Sonnet 4, Claude Sonnet 3.7, Claude Sonnet 3.5, and Claude Opus 3
- 2048 tokens for Claude Haiku 3.5 and Claude Haiku 3

Shorter prompts cannot be cached, even if marked with `cache_control`. Any requests to cache fewer than this number of tokens will be processed without caching. To see if a 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.

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: 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.

What cannot be cached

While most request blocks can be cached, exceptions include:

- Thinking blocks: Cannot be directly marked with `cache_control`, but are cached when they appear in previous assistant turns and are read as input tokens.
- Sub-content blocks (like citations): Cannot be cached directly; cache the top-level block instead.
- Empty text blocks: Cannot be cached.

What invalidates the cache

Modifications to cached content can invalidate some or all of the cache. Changes at each hierarchy level invalidate that level and all subsequent levels.

What changes	Tools cache	System cache	Messages cache	Impact
Tool definitions	✗	✗	✗	Modifying definitions invalidates all
Web search toggle	✓	✗	✗	Changes system prompt
Citations toggle	✓	✗	✗	Changes system prompt
Tool choice	✓	✓	✗	Changes message blocks

What changes	Tools cache	System cache	Messages cache	Impact
Images	✓	✓	✗	Adds/removes images affects message blocks
Thinking parameters	✓	✓	✗	Changes extended thinking settings
Non-tool results in extended thinking	✓	✓	✗	Strips cached thinking blocks

Tracking cache performance

Use the `usage` fields in the API response:

- `cache_creation_input_tokens`: Tokens written during cache creation
- `cache_read_input_tokens`: Tokens retrieved from the cache
- `input_tokens`: Tokens in the user message not from cache

Best practices

- Cache stable, reusable content (system instructions, background info, large contexts)
- Place cached content at the beginning of your prompt
- Use cache breakpoints strategically
- Monitor cache hit rates and adjust

Optimizing for different scenarios

Adjust your caching based on scenario needs, e.g.:

- Conversational agents
- Coding assistants
- Large document processing
- Detailed instructions
- Agent tool use
- Longform content interaction

FAQ

What is the cache lifetime?

Default TTL is 5 minutes, refreshed on each use. For longer, see [1-hour cache TTL](#).

How many cache breakpoints can I use?

Up to 4.

Is prompt caching available for all models?

No, only for Claude Opus 4, Claude Sonnet 4, 3.7, 3.5, Haiku 3.5, 3, and Opus 3.

How does prompt caching work with extended thinking?

System prompts and tools are reused when thinking parameters change. Changes to thinking settings invalidate cached prefixes.

How do I enable prompt caching?

Include at least one `cache_control` breakpoint in your API request.

Can I use prompt caching with other API features?

Yes, but changing images or tool use settings will invalidate the cache.

How does prompt caching affect pricing?

Cache writes cost 25% more; cache hits cost 10%.

Can I manually clear the cache?

No, cached prefixes expire after at least 5 minutes of inactivity automatically.

How to track caching effectiveness?

Monitor `cache_creation_input_tokens` and `cache_read_input_tokens` in responses.

What can break the cache?

See [What invalidates the cache](#).

How does prompt caching handle privacy?

- Cache keys are cryptographically hashed
- Caches are organization-specific
- Maintains context privacy and security

Can I use prompt caching with the Batches API?

Yes, but cache hits are on a best-effort basis. Using a shared prefix with a 1-hour cache can improve cache hit rate.

Why do I see errors like `AttributeError: 'Beta' object has no attribute 'prompt_caching'`?

This occurs with outdated SDK usage. Use the current methods without the beta prefix.

Why do I see `'TypeError: Cannot read properties of undefined (reading 'messages')`?

Likewise, upgrade SDKs and use current API methods.

Capabilities

Building with extended thinking

Extended thinking gives Claude enhanced reasoning capabilities for complex tasks, while providing varying levels of transparency into its step-by-step thought process before it delivers its final answer.

Supported models

Extended thinking is supported in the following models:

- Claude Opus 4 (`claude-opus-4-20250514`)
- Claude Sonnet 4 (`claude-sonnet-4-20250514`)
- Claude Sonnet 3.7 (`claude-3-7-sonnet-20250219`)

API behavior differs across Claude 3.7 and Claude 4 models, but the API shapes remain exactly the same. For more information, see [Differences in thinking across model versions](#).

How extended thinking works

When enabled, Claude creates `thinking` content blocks where it outputs its internal reasoning. It incorporates insights from this reasoning before crafting a final response.

The API response will include `thinking` content blocks, followed by `text` content blocks.

Example of default response format:

```
{
  "content": [
    {
      "type": "thinking",
      "thinking": "Let me analyze this step by step...",
      "signature": "WaUjzkypQ2mUEVM3602TxuC06KN8xyfbJwyem2dw3URve/op91XWH0EBLLqIOMfFG/"
    },
    {
      "type": "text",
      "text": "Based on my analysis..."
    }
  ]
}
```

For more details, see [Messages API Reference](#).

How to use extended thinking

Here's an example of using extended thinking via the Messages API:

```
curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
```

```

--data '{
  "model": "claude-sonnet-4-20250514",
  "max_tokens": 16000,
  "thinking": {
    "type": "enabled",
    "budget_tokens": 10000
  },
  "messages": [
    {
      "role": "user",
      "content": "Are there an infinite number of prime numbers such that n mod 4 ==
    }
  ]
}'

```

To activate extended thinking, include a `thinking` object with `"type": "enabled"` and specify `budget_tokens`. The `budget_tokens` determines the maximum internal reasoning tokens Claude can use. Larger budgets can improve response quality for complex tasks, but Claude may not use the entire budget, especially above 32k tokens.

Note: `budget_tokens` must be less than `max_tokens`. Using interleaved thinking with tools allows exceeding this limit as the total context becomes larger (up to 200k tokens).

Summarized thinking

With enabled extended thinking, Claude's full reasoning process is returned as a summary, providing the benefits of extended thinking while preventing misuse.

Key points:

- Charged for full thinking tokens generated during the original request.
- The billed output tokens do not match the response's token count.
- The initial reasoning output is more verbose.
- Future updates may modify summarization behavior.
- Summarization preserves key ideas with minimal latency, supporting streaming and migration.
- Done by a different model than the main response; it does not see the summarized output.

Claude Sonnet 3.7 continues to return full reasoning output. For access to full reasoning in Claude 4, contact sales@anthropic.com.

Streaming thinking

You can stream extended thinking responses using [Server-Sent Events \(SSE\)](#). When enabled, thinking content arrives via `thinking_delta` events.

Example setup:

```

curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-sonnet-4-20250514",
    "max_tokens": 16000,
    "stream": true,
    "thinking": {

```



```

        "type": "enabled",
        "budget_tokens": 10000
    },
    "messages": [
        {
            "role": "user",
            "content": "What is 27 * 453?"
        }
    ]
}'

```

Partial streamed output:

```

event: message_start
data: {...}

```

```

event: content_block_start
data: {...}

```

```

event: content_block_delta
data: {...}

```

...

Note: Content may arrive in large chunks or token-by-token, especially for reasoning content. This is expected and optimized for performance; future updates aim for smoother streaming.

Extended thinking with tool use

Extended thinking can be combined with tools, allowing Claude to reason about tool selection and results.

Limitations:

1. Tool use with thinking supports only `"tool_choice": {"type": "auto"}` (default) or `"none"`. Other options like `"any"` or specific tool names are incompatible.
2. You must pass back the full, unmodified reasoning blocks during tool use to maintain reasoning continuity.

Example: Passing reasoning with tool results:

```

weather_tool = {
    "name": "get_weather",
    "description": "Get current weather for a location",
    "input_schema": {
        "type": "object",
        "properties": {
            "location": {"type": "string"}
        },
    },
    "required": ["location"]
}

# Initial request
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=16000,
    thinking={"type": "enabled", "budget_tokens": 10000},

```

```

tools=[weather_tool],
messages=[
  {"role": "user", "content": "What's the weather in Paris?"}
]
)

# Extract reasoning and tool use blocks
thinking_block = next((b for b in response.content if b.type == "thinking"), None)
tool_use_block = next((b for b in response.content if b.type == "tool_use"), None)

# Call external weather API here...
weather_data = {"temperature": 88} # Example

# Continue with full reasoning preserved
response2 = client.messages.create(
  model="claude-sonnet-4-20250514",
  max_tokens=16000,
  thinking={"type": "enabled", "budget_tokens": 10000},
  tools=[weather_tool],
  messages=[
    {"role": "user", "content": "What's the weather in Paris?"},
    {"role": "assistant", "content": [thinking_block, tool_use_block]},
    {"role": "user", "content": [{"type": "tool_result", "tool_use_id": tool_use_block}]}
  ]
)

```

The response will include only `text` content, now informed by the tool result and reasoning.

Preserving reasoning blocks

Pass the complete, unmodified reasoning blocks back to the API when using tools. This maintains reasoning flow and conversation integrity. Always include all thinking blocks in multi-turn conversations.

Claude pauses during tool use to process external info. To maintain reasoning continuity, preserve previous thinking blocks in your requests, even if they are flagged or encrypted.

Interleaved thinking

Supports reasoning between tool calls for more nuanced decision-making.

Enabling:

- Add the header `interleaved-thinking-2025-05-14` in API requests.

Limitations:

- The `budget_tokens` can exceed `max_tokens`.
- Supported only for use with tools via the Messages API.
- Only for Claude 4 models with the beta header.
- On third-party platforms, passing this header for unsupported models will fail.

Examples:

- Without interleaved thinking, Claude thinks once at the start, then makes all tool calls, responding without additional reasoning.
- With interleaved thinking, Claude can reason after each tool, chaining steps based on intermediate results.

Refer to [Tool use without interleaved thinking](#) for more.

Extended thinking with prompt caching

Thinking blocks from previous turns are removed from context and don't count against the context window, but they do count for input tokens when cached.

- Modifying thinking parameters (enabled/disabled, budget) invalidates cache.
- Interleaved thinking further complicates cache invalidation.
- When continuing conversations with tools, preserve thinking blocks and cache as needed.

Cache behavior specifics

- Caching occurs only when subsequent requests include tool results.
- Previous thinking blocks are included/excluded based on the presence of new user or assistant blocks.
- Changes in thinking parameters may invalidate message cache.
- When using tools, thinking blocks are cached per turn; redacted or flagged blocks are encrypted but still usable.

Example of caching behavior:

- User asks: "What's the weather in Paris?"
- Response includes reasoning and required tool call. If next user message repeats or continues this context, reasoning blocks might be reused.
- Altering thinking budgets or parameters invalidates cache, prompting new reasoning.

System prompt caching

System prompts are cached and persisted even when thinking parameters change, ensuring consistent behavior across sessions.

Max tokens and context size

Claude models enforce strict `max_tokens` limits. If prompt + `max_tokens` exceeds window size, the API returns an error.

Calculations:

- Fairly account for thoughtful reasoning blocks, which are stripped when not needed, and include current prompts in limit calculations.

Thinking encryption and redaction

Thinking content is encrypted in the `signature` field, used for verification when passing reasoning blocks back. For safety, Claude may redact reasoning, returning encrypted `redacted_thinking` blocks.

Redacted reasoning is decrypted upon return, maintaining conversation flow and safety.

Final Notes

Always pass complete, unmodified reasoning blocks during multi-turn exchanges to ensure reasoning continuity. Be aware of cache invalidation behaviors when changing parameters or using tools.

See further details in related sections on:

- [Preserving thinking blocks](#)
- [Differences across models](#)
- [Prompt engineering](#)
- [Managing tokens](#)

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Streaming Messages - Anthropic

Capabilities

Streaming Messages

When creating a Message, you can set `"stream": true` to incrementally stream the response using [server-sent events](#) (SSE).

Streaming with SDKs

Our [Python](#) and [TypeScript](#) SDKs offer multiple ways of streaming. The Python SDK allows both sync and async streams. See the documentation in each SDK for details.

Python Example

```
import anthropic

client = anthropic.Anthropic()

with client.messages.stream(
    max_tokens=1024,
    messages=[{"role": "user", "content": "Hello"}],
    model="claude-opus-4-20250514"
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)
```

Event types

Each server-sent event includes a named event type and associated JSON data. Each event will use an SSE event name (e.g., `event: message_stop`), and include the matching event `type` in its data.

Event flow

1. `message_start`: contains a `Message` object with empty `content`.
2. A series of content blocks, each with `content_block_start`, one or more `content_block_delta` events, and a `content_block_stop` event. Each content block has an `index`.
3. One or more `message_delta` events, indicating top-level changes.
4. A final `message_stop` event.

The token counts shown in the `usage` field of the `message_delta` event are *cumulative*.

Ping events

Event streams may also include any number of `ping` events.

Error events

We may occasionally send [errors](#) in the event stream. For example, during high usage, you may receive an `overloaded_error`, which would normally correspond to an HTTP 529 in a non-streaming context.

Example error

```
event: error
data: {
  "type": "error",
  "error": {
    "type": "overloaded_error",
    "message": "Overloaded"
  }
}
```

Other events

In accordance with our [versioning policy](#), we may add new event types, and your code should handle unknown event types gracefully.

Content block delta types

Each `content_block_delta` event contains a `delta` of a type that updates the `content` block at a given `index`.

Text delta

A `text` content block delta looks like:

```
{
  "type": "content_block_delta",
  "index": 0,
  "delta": {
    "type": "text_delta",
    "text": "ello frien"
  }
}
```

Input JSON delta

The deltas for `tool_use` content blocks correspond to updates for the `input` field of the block. To support maximum granularity, the deltas are *partial JSON strings*, whereas the final `tool_use.input` is always an *object*.

You can accumulate the string deltas and parse the JSON once received a `content_block_stop` event, using a library like [Pydantic](#) or via our [SDKs](#).

Example

```
{
  "type": "content_block_delta",
  "index": 1,
  "delta": {
    "type": "input_json_delta",
    "partial_json": "{\"location\": \"San Fra\"}"
  }
}
```

Note: Models currently only support emitting one complete key and value at a time for `input`. This may cause delays while the model is working, but allows support for finer granularity in future models.

Thinking delta

When using [extended thinking](#) with streaming, you'll receive thinking content via `thinking_delta` events, which correspond to the `thinking` field of the `thinking` blocks.

A signature delta is sent just before the `content_block_stop` event for verifying integrity.

Example thinking delta:

```
{
  "type": "content_block_delta",
  "index": 0,
  "delta": {
    "type": "thinking_delta",
    "thinking": "Let me solve this step by step:\n\n1. First break down 27 * 453"
  }
}
```

Signature delta example:

```
{
  "event": "content_block_delta",
  "data": {
    "type": "signature_delta",
    "signature": "EqQBCgIYAhIM1gbcDa9GJwZA2b3hGgxBdjrkzLoky3dl1pkiM0Yds..."
  }
}
```

Full HTTP Stream response

We recommend using [client SDKs](#) for streaming, but if building directly, you'll need to handle events.

A stream response includes:

1. `message_start` event
2. Multiple content blocks:
 - `content_block_start`
 - Several `content_block_delta`
 - `content_block_stop`
3. `message_delta`
4. `message_stop`

Basic stream request

```
curl https://api.anthropic.com/v1/messages \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --data '{
    "model": "claude-opus-4-20250514",
    "messages": [{"role": "user", "content": "Hello"}],
    "max_tokens": 256,
```

```
    "stream": true
  }'
```

With tool use

Request example:

```
curl https://api.anthropic.com/v1/messages \
  --header "content-type: application/json" \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --data '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "tools": [
      {
        "name": "get_weather",
        "description": "Get the current weather in a given location",
        "input_schema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            }
          }
        },
        "required": ["location"]
      }
    ],
    "tool_choice": {"type": "any"},
    "messages": [{"role": "user", "content": "What is the weather like in San Franc"},
    "stream": true
  }'
```

Response involves events like:

- `message_start`
- `content_block_start`, `content_block_delta`, `content_block_stop`
- updates with `message_delta`
- final `message_stop`

With web search tool use

Example request:

```
curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-opus-4-20250514",
    "messages": [{"role": "user", "content": "What's the current weather in New Yor"},
    "stream": true,
    "content_block": {
      "type": "server_tool_use",

```

```
    "id": "srvtoolu_014hJH82Qum7Td6UV8gDXThB",
    "name": "web_search",
    "input": {}
  }
}'
```

Response will include:

- `message_start`
- `content_block_start`, `content_block_delta`, `content_block_stop` with search results, e.g., [Weather in NYC - detailed forecast](#)
- Final `message_stop`

Additional

Was this page helpful? [Yes](#), [No](#)

Links:

- [Extended thinking](#)
- [Batch processing](#)

Follow us on:

- [X](#)
- [LinkedIn](#)
- [Discord](#)

Batch processing - Anthropic

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

[Batch processing - Anthropic](#)

Citations - Anthropic

How citations work

Integrate citations with Claude in these steps:

1. Provide document(s) and enable citations
 - Include documents in any of the supported formats: [PDFs](#), [plain text](#), or [custom content](#) documents
 - Set `citations.enabled=true` on each of your documents. Currently, citations must be enabled on all or none of the documents within a request.
 - Note that only text citations are currently supported and image citations are not yet possible.
2. Documents get processed
 - Document contents are “chunked” in order to define the minimum granularity of possible citations. For example, sentence chunking would allow Claude to cite a single sentence or chain together multiple consecutive sentences to cite a paragraph (or longer)!
 - **For PDFs:** Text is extracted as described in [PDF Support](#) and content is chunked into sentences. Citing images from PDFs is not currently supported.
 - **For plain text documents:** Content is chunked into sentences that can be cited from.
 - **For custom content documents:** Your provided content blocks are used as-is and no further chunking is done.
3. Claude provides cited response
 - Responses may now include multiple text blocks where each text block can contain a claim that Claude is making and a list of citations that support the claim.
 - Citations reference specific locations in source documents. The format of these citations depend on the document type:
 - **For PDFs:** Include the page number range (1-indexed).
 - **For plain text:** Include the character index range (0-indexed).
 - **For custom content:** Include the content block index range (0-indexed).
 - Document indices are 0-indexed according to the list of all documents in your original request.

Citable vs non-citable content

- Text within a document's `source` can be cited.
- `title` and `context` are optional fields, passed to the model but not used as cited content.
- `title` is limited in length; `context` can store document metadata as text or stringified JSON.

Citation indices

- Document indices are 0-indexed from the list of all document content blocks (spanning across all messages).
- Character indices are 0-indexed with exclusive end indices.
- Page numbers are 1-indexed with exclusive end page numbers.
- Content block indices are 0-indexed with exclusive end indices from the `content` list in custom content documents.

Token costs

- Enabling citations increases input tokens slightly due to system prompt additions and document chunking.
- Citation output tokens are efficient: citations are in a standardized format parsed into cited text and location indices.
- The `cited_text` field does not count towards output tokens.
- When passed in subsequent turns, `cited_text` is also not counted towards input tokens.

Feature compatibility

Citations work with other API features including [prompt caching](#), [token counting](#), and [batch processing](#).

Using Prompt Caching with Citations

- Citations and prompt caching can be used together effectively.
- Citation blocks generated in responses cannot be cached directly, but source documents can be cached.
- Apply `cache_control` to your top-level document content blocks for performance optimization.

```
import anthropic

client = anthropic.Anthropic()

# Long document content (e.g., technical documentation)
long_document = "This is a very long document with thousands of words..." + " ... " * 1000

response = client.messages.create(
    model="claude-opus-4-20250514",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "document",
                    "source": {
                        "type": "text",
                        "media_type": "text/plain",
                        "data": long_document
                    },
                    "citations": {"enabled": True},
                    "cache_control": {"type": "ephemeral"} # Cache the document content
                },
                {
                    "type": "text",
                    "text": "What does this document say about API features?"
                }
            ]
        }
    ]
)
```

- **In this example:**
 - The document content is cached via `cache_control`.
 - Citations are enabled on the document.
 - Claude can generate responses with citations benefits from cached content.
 - Subsequent requests with the same document will use the cache.

Document Types

Choosing a document type

Supported formats for citations:

Type	Best for	Chunking	Citation format
Plain text	Simple text documents, prose	Sentence	Character indices (0-based)
PDF	PDF files with text content	Sentence	Page numbers (1-based)
Custom content	Lists, transcripts, special formatting, granular citations	No	Block indices (0-based)

Note: `.csv`, `.xlsx`, `.docx`, `.md`, `.txt` files are not supported as document blocks. Convert to plain text and include directly in message content.

Plain text documents

- Chunked into sentences automatically.
- Provide inline or by reference using `file_id`:
 - Inline text
 - Files API

Example:

```
{
  "type": "document",
  "source": {
    "type": "text",
    "media_type": "text/plain",
    "data": "Plain text content..."
  },
  "title": "Document Title",
  "context": "Context about the document that will not be cited from",
  "citations": {"enabled": true}
}
```

PDF documents

- Provide as base64 or by `file_id`.
- Extracted text chunked into sentences.
- Image scans are not citable.

Example:

```
{
  "type": "document",
  "source": {
    "type": "base64",
    "media_type": "application/pdf",
    "data": "base64-encoded-pdf-data"
  },
  "title": "Document Title",
  "citations": {"enabled": true}
}
```


Custom content documents

- No additional chunking; use content blocks as is.
- Example:

```
{
  "type": "document",
  "source": {
    "type": "content",
    "content": [
      {"type": "text", "text": "First chunk"},
      {"type": "text", "text": "Second chunk"}
    ]
  },
  "title": "Document Title",
  "citations": {"enabled": true}
}
```

- Citation example:

```
{
  "type": "content_block_location",
  "cited_text": "The exact text being cited",
  "document_index": 0,
  "start_block_index": 0,
  "end_block_index": 1
}
```

Response Structure

Responses include text blocks with citations:

```
{
  "content": [
    {
      "type": "text",
      "text": "According to the document, "
    },
    {
      "type": "text",
      "text": "the grass is green",
      "citations": [
        {
          "type": "char_location",
          "cited_text": "The grass is green.",
          "document_index": 0,
          "start_char_index": 0,
          "end_char_index": 20
        }
      ]
    }
  ]
}
```

Streaming Support

- Adds `citations_delta` containing citation info for real-time updates.
- Example events:

```
event: message_start
```

```
data: {"type": "message_start"}
```

```
event: content_block_start
```

```
data: {"type": "content_block_start", "index": 0}
```

```
event: content_block_delta
```

```
data: {  
  "type": "content_block_delta",  
  "index": 0,  
  "delta": {"type": "citations_delta",  
            "citation": {  
              "type": "char_location",  
              "cited_text": "...",  
              "document_index": 0  
            }  
          }  
}
```

```
event: content_block_stop
```

```
data: {"type": "content_block_stop", "index": 0}
```

```
event: message_stop
```

```
data: {"type": "message_stop"}
```

Multilingual support - Anthropic

Overview

Claude excels at tasks across multiple languages, maintaining strong cross-lingual performance relative to English.

Performance data

Below are the zero-shot chain-of-thought evaluation scores for Claude 4, Claude 3.7 Sonnet and Claude 3.5 models across different languages, shown as a percent relative to English performance (100%):

Language	Claude Opus 4 ¹	Claude Sonnet 4 ¹	Claude Sonnet 3.7 ¹	Claude Sonnet 3.5 v2	Claude Haiku 3.5
English (baseline, fixed to 100%)	100%	100%	100%	100%	100%
Spanish	98.0%	97.5%	97.6%	96.9%	94.6%
Portuguese (Brazil)	97.3%	97.2%	97.3%	96.0%	94.6%
Italian	97.5%	97.3%	97.2%	95.6%	95.0%
French	97.7%	97.1%	96.9%	96.2%	95.3%
Indonesian	97.2%	96.2%	96.3%	94.0%	91.2%
German	97.1%	94.7%	96.2%	94.0%	92.5%
Arabic	96.9%	96.1%	95.4%	92.5%	84.7%
Chinese (Simplified)	96.7%	95.9%	95.3%	92.8%	90.9%
Korean	96.4%	95.9%	95.2%	92.8%	89.1%
Japanese	96.2%	95.6%	95.0%	92.7%	90.8%
Hindi	96.7%	95.8%	94.2%	89.3%	80.1%
Bengali	95.2%	94.4%	92.4%	85.9%	72.9%
Swahili	89.5%	87.1%	89.2%	83.9%	64.7%
Yoruba	78.9%	76.4%	76.7%	64.9%	46.1%

¹ With [extended thinking](#).

These metrics are based on [MMLU \(Massive Multitask Language Understanding\)](#) English test sets that were translated into 14 additional languages by professional human translators, as documented in [OpenAI's simple-evals repository](#). The use of human translators for this evaluation ensures high-quality translations, particularly important for languages with fewer digital resources.

Best practices

When working with multilingual content:

- 1. Provide clear language context:** While Claude can detect the target language automatically, explicitly stating the desired input/output language improves reliability. For enhanced fluency, you can prompt Claude to use “idiomatic speech as if it were a native speaker.”
- 2. Use native scripts:** Submit text in its native script rather than transliteration for optimal results.
- 3. Consider cultural context:** Effective communication often requires cultural and regional awareness beyond pure translation.

We also suggest following our general [prompt engineering guidelines](#) to better improve Claude's performance.

Language support considerations

- Claude processes input and generates output in most world languages that use standard Unicode characters.
- Performance varies by language, with particularly strong capabilities in widely-spoken languages.
- Even in languages with fewer digital resources, Claude maintains meaningful capabilities.

[Prompt Engineering Guide](#)
[Prompt Library](#)

Was this page helpful?

Yes | No

Citations

[Anthropic Privacy Policy](#)
[Security and compliance](#)

On this page

- [Overview](#)
- [Performance data](#)
- [Best practices](#)
- [Language support considerations](#)

Token counting - Anthropic

Capabilities

Token counting

Token counting enables you to determine the number of tokens in a message before sending it to Claude, helping you make informed decisions about your prompts and usage. With token counting, you can

- Proactively manage rate limits and costs
- Make smart model routing decisions
- Optimize prompts to be a specific length

How to count message tokens

The [token counting](#) endpoint accepts the same structured list of inputs for creating a message, including support for system prompts, [tools](#), [images](#), and [PDFs](#). The response contains the total number of input tokens.

The token count should be considered an **estimate**. In some cases, the actual number of input tokens used when creating a message may differ by a small amount.

Supported models

The token counting endpoint supports the following models:

- Claude Opus 4
- Claude Sonnet 4
- Claude Sonnet 3.7
- Claude Sonnet 3.5
- Claude Haiku 3.5
- Claude Haiku 3
- Claude Opus 3

Count tokens in basic messages

Python

```
import anthropic

client = anthropic.Anthropic()

response = client.messages.count_tokens(
    model="claude-opus-4-20250514",
    system="You are a scientist",
    messages=[
        {
            "role": "user",
            "content": "Hello, Claude"
        }
    ]
)
```

```
print(response.json())
```

Response JSON

```
{
  "input_tokens": 14
}
```

Count tokens in messages with tools

Python

```
import anthropic

client = anthropic.Anthropic()

response = client.messages.count_tokens(
    model="claude-opus-4-20250514",
    tools=[
        {
            "name": "get_weather",
            "description": "Get the current weather in a given location",
            "input_schema": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g., San Francisco, CA"
                    }
                },
                "required": ["location"]
            }
        }
    ],
    messages=[
        {
            "role": "user",
            "content": "What's the weather like in San Francisco?"
        }
    ]
)

print(response.json())
```

Response JSON

```
{
  "input_tokens": 403
}
```

Count tokens in messages with images

Shell

```
#!/bin/sh
```

```
IMAGE_URL="https://upload.wikimedia.org/wikipedia/commons/a/a7/Camponotus_flavomargina"
IMAGE_MEDIA_TYPE="image/jpeg"
IMAGE_BASE64=$(curl "$IMAGE_URL" | base64)
```

```
curl https://api.anthropic.com/v1/messages/count_tokens \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-opus-4-20250514",
    "messages": [
      {
        "role": "user",
        "content": [
          {
            "type": "image",
            "source": {
              "type": "base64",
              "media_type": "'$IMAGE_MEDIA_TYPE'",
              "data": "'$IMAGE_BASE64'"
            }
          },
          {
            "type": "text",
            "text": "Describe this image"
          }
        ]
      }
    ]
  }'
```

Response JSON

```
{
  "input_tokens": 1551
}
```

Count tokens in messages with extended thinking

Shell

```
curl https://api.anthropic.com/v1/messages/count_tokens \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "content-type: application/json" \
  --header "anthropic-version: 2023-06-01" \
  --data '{
    "model": "claude-opus-4-20250514",
    "thinking": {
      "type": "enabled",
      "budget_tokens": 16000
    },
    "messages": [
      {
        "role": "user",
        "content": "Are there an infinite number of prime numbers such that n mod 4"
      },
    ]
  }'
```

```
{
  "role": "assistant",
  "content": [
    {
      "type": "thinking",
      "thinking": "This is a nice number theory question. Lets think about it",
      "signature": "EuYBckQYAiJAgCs1le6/Pol5Z4/JMomVOouGrWdhYNsh3ukzUECbB6iWrS"
    },
    {
      "type": "text",
      "text": "Yes, there are infinitely many prime numbers p such that p mod"
    }
  ]
},
{
  "role": "user",
  "content": "Can you write a formal proof?"
}
]
```

Response JSON

```
{
  "input_tokens": 88
}
```

Count tokens in messages with PDFs

Shell

```
curl https://api.anthropic.com/v1/messages/count_tokens \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "content-type: application/json" \
  --header "anthropic-version: 2023-06-01" \
  --data '{
    "model": "claude-opus-4-20250514",
    "messages": [
      {
        "role": "user",
        "content": [
          {
            "type": "document",
            "source": {
              "type": "base64",
              "media_type": "application/pdf",
              "data": "'$(base64 -i document.pdf)'"
            }
          },
          {
            "type": "text",
            "text": "Please summarize this document."
          }
        ]
      }
    ]
  }'
```



```
]
}'
```

Response JSON

```
{
  "input_tokens": 2188
}
```

Pricing and rate limits

Token counting is **free to use** but subject to requests per minute rate limits based on your [usage tier](#). If you need higher limits, contact sales through the [Anthropic Console](#).

Usage tier Requests per minute (RPM)

1	100
2	2,000
3	4,000
4	8,000

Token counting and message creation have separate and independent rate limits — usage of one does not count against the limits of the other.

FAQ

Does token counting use prompt caching?

No, token counting provides an estimate without using caching logic. While you may provide `cache_control` blocks in your token counting request, prompt caching only occurs during actual message creation.

Was this page helpful?

[Yes](#) / [No](#)

On this page

- [How to count message tokens](#)
- [Supported models](#)
- [Count tokens in basic messages](#)
- [Count tokens in messages with tools](#)
- [Count tokens in messages with images](#)
- [Count tokens in messages with extended thinking](#)
- [Count tokens in messages with PDFs](#)
- [Pricing and rate limits](#)
- [FAQ](#)

Embeddings - Anthropic

Capabilities

Embeddings

[Copy page](#)

Text embeddings are numerical representations of text that enable measuring semantic similarity. This guide introduces embeddings, their applications, and how to use embedding models for tasks like search, recommendations, and anomaly detection.

Before implementing embeddings

When selecting an embeddings provider, there are several factors you can consider depending on your needs and preferences:

- **Dataset size & domain specificity:** size of the model training dataset and its relevance to the domain you want to embed. Larger or more domain-specific data generally produces better in-domain embeddings
- **Inference performance:** embedding lookup speed and end-to-end latency. This is a particularly important consideration for large scale production deployments
- **Customization:** options for continued training on private data, or specialization of models for very specific domains. This can improve performance on unique vocabularies

How to get embeddings with Anthropic

Anthropic does not offer its own embedding model. One embeddings provider that has a wide variety of options and capabilities encompassing all of the above considerations is Voyage AI.

Voyage AI makes state-of-the-art embedding models and offers customized models for specific industry domains such as finance and healthcare, or bespoke fine-tuned models for individual customers.

The rest of this guide is for Voyage AI, but we encourage you to assess a variety of embeddings vendors to find the best fit for your specific use case.

Available Models

Voyage recommends using the following text embedding models:

Model	Context Length	Embedding Dimension	Description
voyage-3-large	32,000	1024 (default), 256, 512, 2048	The best general-purpose and multilingual retrieval quality. See blog post for details.
voyage-3.5	32,000	1024 (default), 256, 512, 2048	Optimized for general-purpose and multilingual retrieval quality. See blog post for details.
voyage-3.5-lite	32,000	1024 (default), 256, 512, 2048	Optimized for latency and cost. See blog post for details.
voyage-code-3	32,000	1024 (default), 256, 512, 2048	Optimized for <i>code</i> retrieval. See blog post for details.

Model	Context Length	Embedding Dimension	Description
voyage-finance-2	32,000	1024	Optimized for <i>finance</i> retrieval and RAG. See blog post for details.
voyage-law-2	16,000	1024	Optimized for <i>legal</i> and <i>long-context</i> retrieval and RAG. Also improved performance across all domains. See blog post for details.

Additionally, the following multimodal embedding models are recommended:

Model	Context Length	Embedding Dimension	Description
voyage-multimodal-3	32,000	1024	Rich multimodal embedding model that can vectorize interleaved text and content-rich images, such as screenshots of PDFs, slides, tables, figures, and more. See blog post for details.

Need help deciding which text embedding model to use? Check out the [FAQ](#).

Getting started with Voyage AI

To access Voyage embeddings:

1. Sign up on Voyage AI's website
2. Obtain an API key
3. Set the API key as an environment variable for convenience:

```
export VOYAGE_API_KEY="<your secret key>"
```

You can obtain the embeddings by either using the official [voyageai Python package](#) or HTTP requests, as described below.

Voyage Python library

The `voyageai` package can be installed using:

```
pip install -U voyageai
```

Then, you can create a client object and start using it to embed your texts:

```
import voyageai

vo = voyageai.Client()
# This will automatically use the environment variable VOYAGE_API_KEY.
# Alternatively, you can use vo = voyageai.Client(api_key="&lt;your secret key&gt;")

texts = ["Sample text 1", "Sample text 2"]
result = vo.embed(texts, model="voyage-3.5", input_type="document")
print(result.embeddings[0])
print(result.embeddings[1])
```

`result.embeddings` will be a list of two embedding vectors, each containing 1024 floating-point numbers. After running the above code, the two embeddings will be printed on the screen:

```
[-0.013131560757756233, 0.019828535616397858, ...] # embedding for "Sample text 1"
[-0.0069352793507277966, 0.020878976210951805, ...] # embedding for "Sample text 2"
```

When creating the embeddings, you can specify other arguments to the `embed()` function.

For more information on the Voyage Python package, see [the Voyage documentation](#).

Voyage HTTP API

You can also get embeddings by requesting Voyage HTTP API. For example, send an HTTP request via `curl`:

```
curl https://api.voyageai.com/v1/embeddings \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $VOYAGE_API_KEY" \
-d '{
  "input": ["Sample text 1", "Sample text 2"],
  "model": "voyage-3.5"
}'
```

The response will be a JSON object containing the embeddings and token usage:

```
{
  "object": "list",
  "data": [
    {
      "embedding": [-0.013131560757756233, 0.019828535616397858, ...],
      "index": 0
    },
    {
      "embedding": [-0.0069352793507277966, 0.020878976210951805, ...],
      "index": 1
    }
  ],
  "model": "voyage-3.5",
  "usage": {
    "total_tokens": 10
  }
}
```

For more information on the Voyage HTTP API, see [the Voyage documentation](#).

AWS Marketplace

Voyage embeddings are available on [AWS Marketplace](#). Instructions for accessing Voyage on AWS are [here](#).

Quickstart example

Now that we know how to get embeddings, let's see a brief example.

Suppose we have a small corpus of six documents to retrieve from:

```
documents = [
  "The Mediterranean diet emphasizes fish, olive oil, and vegetables, believed to re
  "Photosynthesis in plants converts light energy into glucose and produces essentia
  "20th-century innovations, from radios to smartphones, centered on electronic adva
  "Rivers provide water, irrigation, and habitat for aquatic species, vital for ecos
  "Apple's conference call to discuss fourth fiscal quarter results and business upd
  "Shakespeare's works, like 'Hamlet' and 'A Midsummer Night's Dream,' endure in lit
]
```

We will first use Voyage to convert each of them into an embedding vector:

```

import voyageai

vo = voyageai.Client()

# Embed the documents
doc_embds = vo.embed(
    documents,
    model="voyage-3.5",
    input_type="document"
).embeddings

```

The embeddings will allow us to do semantic search / retrieval in the vector space. Given a query:

```
query = "When is Apple's conference call scheduled?"
```

We convert it into an embedding, and conduct a nearest neighbor search to find the most relevant document:

```

import numpy as np

# Embed the query
query_embd = vo.embed(
    [query],
    model="voyage-3.5",
    input_type="query"
).embeddings[0]

# Compute the similarity
# Voyage embeddings are normalized, so dot-product equals cosine similarity
similarities = np.dot(doc_embds, query_embd)

retrieved_id = np.argmax(similarities)
print(documents[retrieved_id])

```

Note: Use `input_type="document"` and `input_type="query"` for embedding the document and query, respectively. More details are [here](#).

The output would be the fifth document, which is indeed the most relevant:

```
Apple's conference call to discuss fourth fiscal quarter results and business updates
```

If you want a detailed set of cookbooks on how to do RAG with embeddings, including vector databases, check out our [RAG cookbook](#).

FAQ

Why do Voyage embeddings have superior quality?

Embedding models rely on powerful neural networks to capture and compress semantic context, similar to generative models. Voyage's team of experienced AI researchers optimizes every component of the embedding process, including:

- Model architecture
- Data collection
- Loss functions
- Optimizer selection

Learn more about Voyage's technical approach on their [blog](#).

What embedding models are available and which should I use?

For general-purpose embedding, we recommend:

- `voyage-3-large` : Best quality
- `voyage-3.5-lite` : Lowest latency and cost
- `voyage-3.5` : Balanced performance with superior retrieval quality at a competitive price point

For retrieval, use the `input_type` parameter to specify whether the text is a query or document type.

Domain-specific models:

- Legal tasks: `voyage-law-2`
- Code and programming documentation: `voyage-code-3`
- Finance-related tasks: `voyage-finance-2`

Which similarity function should I use?

You can use Voyage embeddings with either dot-product similarity, cosine similarity, or Euclidean distance. An explanation about embedding similarity is [here](#).

Voyage AI embeddings are normalized to length 1, which means that:

- Cosine similarity is equivalent to dot-product similarity, while the latter can be computed more quickly.
- Cosine similarity and Euclidean distance will produce the same rankings.

What is the relationship between characters, words, and tokens?

Please see [this page](#).

When and how should I use the `input_type` parameter?

For retrieval tasks and use cases (e.g., RAG), it's recommended to specify `input_type` as either `query` or `document`. Do not omit it or set it to `None`.

Specifying the input type creates better dense vector representations, leading to improved retrieval quality.

When using `input_type`, special prompts are prepended to the input text:

■ Prompts associated with `input_type`

- For a query: "Represent the query for retrieving supporting documents: "
- For a document: "Represent the document for retrieval: "
- Example:
 - When `input_type="query"`, "When is Apple's conference call scheduled?" becomes "Represent the query for retrieving supporting documents: When is Apple's conference call scheduled?"
 - When `input_type="document"`, the description of an article is similarly prefixed.

`voyage-large-2-instruct` is trained to be responsive to additional instructions.

What quantization options are available?

Quantization reduces storage and computation costs of embeddings by converting high-precision values into lower-precision formats like 8-bit integers or 1-bit binary values.

Supported options:

- `float` : Default, highest precision.
- `int8` and `uint8` : 8-bit integer representations.
- `binary` and `ubinary` : Bit-packed, quantized representations, with specific packing examples.

How can I truncate Matryoshka embeddings?

Matryoshka embeddings support multiple output dimensions, and can be truncated by keeping the leading subset.

Example to truncate from 1024 to 256 dimensions:

```
import voyageai
import numpy as np

def embd_normalize(v: np.ndarray) -> np.ndarray:
    """Normalize rows of a 2D numpy array to unit vectors."""
    row_norms = np.linalg.norm(v, axis=1, keepdims=True)
    if np.any(row_norms == 0):
        raise ValueError("Cannot normalize rows with a norm of zero.")
    return v / row_norms

vo = voyageai.Client()

# Generate voyage-code-3 vectors (1024-dim)
embds = vo.embed(['Sample text 1', 'Sample text 2'], model='voyage-code-3').embeddings

# Set target shorter dimension
short_dim = 256

# Resize and normalize vectors
resized_embd = embd_normalize(np.array(embds)[:, :short_dim]).tolist()
```

Pricing

Visit Voyage's [pricing page](#) for the most up-to-date pricing details.

Was this page helpful?

- Yes
- No

[Token counting](#) | [Vision](#)

Follow us: [X](#), [LinkedIn](#), [Discord](#)

On this page

- [Before implementing embeddings](#)
- [How to get embeddings with Anthropic](#)
- [Available Models](#)
- [Getting started with Voyage AI](#)
- [Voyage Python library](#)
- [Voyage HTTP API](#)
- [AWS Marketplace](#)

- [Quickstart example](#)
- [FAQ](#)
- [Pricing](#)

Vision - Anthropic

Introduction

The Claude 3 and 4 families of models come with new vision capabilities that allow Claude to understand and analyze images, opening up exciting possibilities for multimodal interaction.

This guide describes how to work with images in Claude, including best practices, code examples, and limitations to keep in mind.

How to use vision

Use Claude's vision capabilities via:

- [claude.ai](#). Upload an image like you would a file, or drag and drop an image directly into the chat window.
 - [Console Workbench](#). If you select a model that accepts images (Claude 3 and 4 models only), a button to add images appears at the top right of every User message block.
 - [API request](#). See the examples in this guide.
-

Before you upload

Basics and Limits

You can include multiple images in a single request (up to 20 for [claude.ai](#) and 100 for API requests). Claude will analyze all provided images when formulating its response. This can be helpful for comparing or contrasting images.

- Images larger than 8000x8000 px will be rejected.
- For up to 20 images in one API request, maximum size per image is ~2000x2000 px.

Evaluate image size

For optimal performance, resize images before uploading if they are too large. If your image's long edge exceeds 1568 pixels, or the image size is over ~1,600 tokens, it will first be scaled down while preserving aspect ratio until within size limits.

Large images increase latency of `time-to-first-token`, without improving model performance. Very small images under 200 pixels on any edge may degrade performance.

Recommendation: Resize images to no more than 1.15 megapixels (e.g., 1568 pixels in both dimensions).

Maximum image sizes (Claude Sonnet 3.7)

Aspect ratio	Image size
1:1	1092x1092 px
3:4	951x1268 px
2:3	896x1344 px
9:16	819x1456 px

Aspect ratio Image size
1:2 784x1568 px

Calculate image costs

Each image counts toward your token usage. To estimate cost:

- Multiply the approximate number of image tokens by the per-token price [here](#).

Estimate tokens with:

$\text{tokens} = (\text{width_px} * \text{height_px}) / 750$

Examples:

Image size	Approximate # Tokens	Cost per image	Cost for 1K images
200x200 px (0.04 MP)	~54	~\$0.00016	~\$0.16
1000x1000 px (1 MP)	~1334	~\$0.004	~\$4.00
1092x1092 px (1.19 MP)	~1590	~\$0.0048	~\$4.80

Ensuring image quality

For best results:

- Use supported formats: JPEG, PNG, GIF, WebP.
- Ensure clarity, avoid blurriness or pixelation.
- Text should be legible and not too small to avoid cropping critical visual info.

Prompt examples

Various prompting techniques used for text work also apply to images, with the preferred structure being images before text for clarity.

About the prompt examples

You can provide images in three ways:

1. Base64-encoded in `image` blocks
2. URL references
3. Files API (upload once, reuse multiple times)

Base64 example

```
# Encode image in bash:  
BASE64_IMAGE_DATA=$(curl -s "https://upload.wikimedia.org/wikipedia/commons/a/a7/Campo
```

API example with base64

Using Base64

```
{  
  "model": "claude-sonnet-4-20250514",  
  "max_tokens": 1024,  
  "messages": [  
    {  
      "role": "user",  
      "content": "  
        image:  
        data:image/jpeg;base64:  
        "BASE64_IMAGE_DATA"  
      }  
    ]  
  }  
}
```

```

{
  "role": "user",
  "content": [
    {
      "type": "image",
      "source": {
        "type": "base64",
        "media_type": "image/jpeg",
        "data": "'$BASE64_IMAGE_DATA'"
      }
    },
    {
      "type": "text",
      "text": "Describe this image."
    }
  ]
}

```

Using URL

```

{
  "model": "claude-sonnet-4-20250514",
  "max_tokens": 1024,
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "image",
          "source": {
            "type": "url",
            "url": "https://upload.wikimedia.org/wikipedia/commons/a/a7/Camponotus_fla"
          }
        },
        {
          "type": "text",
          "text": "Describe this image."
        }
      ]
    }
  ]
}

```

Files API example

Upload images with:

```

curl -X POST https://api.anthropic.com/v1/files \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "anthropic-beta: files-api-2025-04-14" \
  -F "file=@image.jpg"

```

Use `file_id` in messages:

```
{
  "role": "user",
  "content": [
    {
      "type": "image",
      "source": {
        "type": "file",
        "file_id": "file_abc123"
      }
    },
    {
      "type": "text",
      "text": "Describe this image."
    }
  ]
}
```

FAQ: Image Files and Metadata

- Supported types: JPEG, PNG, GIF, WebP
 - Max size: 5MB per image via API; 10MB at claude.ai
 - Up to 100 images (API), 20 per turn (claude.ai)
 - Claude does not parse image metadata
 - Uploads are ephemeral; images are deleted after processing
 - No image generation or editing capabilities
-

Dive deeper into vision

Explore resources:

- [Multimodal cookbook](#)
 - Examples on [getting started with images](#)
 - [API documentation](#)
-

Limitations

- Cannot identify specific people in images
- May hallucinate or mistake low-quality or small images
- Limited spatial reasoning (e.g., analog clocks, chess positions)
- Only approximate counts of objects
- Cannot determine AI-generated images
- Won't process explicit or inappropriate images
- Not suited for complex medical diagnostics

Always verify Claude's interpretations, especially in critical cases.

FAQ

- Supported formats: JPEG, PNG, GIF, WebP
- URLs are processable with `"type": "url"` source
- Max file size: 5MB API, 10MB claude.ai
- Up to 100 images per request, 20 per turn

- Claude does not read image metadata
 - Uploaded images are transient; cannot be deleted manually
 - For privacy details, see our [privacy policy](#)
 - Cannot generate or edit images
-

Dive deeper into vision

Start building with resources such as:

- [Multimodal cookbook](#)
- [API reference](#)

For support, contact [support](#) or join the [Discord community](#).

PDF support - Anthropic

Capabilities

PDF support

Process PDFs with Claude. Extract text, analyze charts, and understand visual content from your documents.

You can now ask Claude about any text, pictures, charts, and tables in PDFs you provide. Some sample use cases:

- Analyzing financial reports and understanding charts/tables
- Extracting key information from legal documents
- Translation assistance for documents
- Converting document information into structured formats

Before you begin

Check PDF requirements

Claude works with any standard PDF. However, you should ensure your request size meet these requirements when using PDF support:

Requirement	Limit
Maximum request size	32MB
Maximum pages per request	100
Format	Standard PDF (no passwords/encryption)

Please note that both limits are on the entire request payload, including any other content sent alongside PDFs.

Since PDF support relies on Claude's vision capabilities, it is subject to the same [limitations and considerations](#) as other vision tasks.

Supported platforms and models

PDF support is currently supported via direct API access and Google Vertex AI on:

- Claude Opus 4 (`claude-opus-4-20250514`)
- Claude Sonnet 4 (`claude-sonnet-4-20250514`)
- Claude Sonnet 3.7 (`claude-3-7-sonnet-20250219`)
- Claude Sonnet 3.5 (`claude-3-5-sonnet-20241022` , `claude-3-5-sonnet-20240620`)
- Claude Haiku 3.5 (`claude-3-5-haiku-20241022`)

This functionality will be supported on Amazon Bedrock soon.

For non-PDF files like `.csv` , `.xlsx` , `.docx` , `.md` , or `.txt` files, see [Working with other file formats](#).

Process PDFs with Claude

Send your first PDF request

Let's start with a simple example using the Messages API. You can provide PDFs to Claude in three ways:

1. As a URL reference to a PDF hosted online
2. As a base64-encoded PDF in `document` content blocks
3. By a `file_id` from the [Files API](#)

Option 1: URL-based PDF document

The simplest approach is to reference a PDF directly from a URL:

```
curl https://api.anthropic.com/v1/messages \
-H "content-type: application/json" \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-d '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "document",
          "source": {
            "type": "url",
            "url": "https://assets.anthropic.com/m/1cd9d098ac3e6467/original/Claude-3-Model-Card-20250314.pdf"
          }
        },
        {
          "type": "text",
          "text": "What are the key findings in this document?"
        }
      ]
    }
  ]
}'
```

Option 2: Base64-encoded PDF document

If you need to send PDFs from your local system or when a URL isn't available:

```
# Method 1: Fetch and encode a remote PDF
curl -s "https://assets.anthropic.com/m/1cd9d098ac3e6467/original/Claude-3-Model-Card-20250314.pdf" | base64 -w 0 > pdf_base64.txt

# Method 2: Encode a local PDF file
# base64 document.pdf | tr -d '\n' > pdf_base64.txt

# Create a JSON request file using the pdf_base64.txt content
jq -n --rawfile PDF_BASE64 pdf_base64.txt '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "messages": [
    {
```

```

    "role": "user",
    "content": [
      {
        "type": "document",
        "source": {
          "type": "base64",
          "media_type": "application/pdf",
          "data": $PDF_BASE64
        }
      },
      {
        "type": "text",
        "text": "What are the key findings in this document?"
      }
    ]
  }
]
}' > request.json

```

```

# Send the API request
curl https://api.anthropic.com/v1/messages \
-H "content-type: application/json" \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-d @request.json

```

Option 3: Files API

For PDFs you'll use repeatedly, or when you want to avoid encoding overhead, use the [Files API](#):

```

# Upload your PDF
curl -X POST https://api.anthropic.com/v1/files \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-H "anthropic-beta: files-api-2025-04-14" \
-F "file=@document.pdf"

```

```

# Then use the returned file_id in your message
curl https://api.anthropic.com/v1/messages \
-H "content-type: application/json" \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-H "anthropic-beta: files-api-2025-04-14" \
-d '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "document",
          "source": {
            "type": "file",
            "file_id": "file_abc123"
          }
        }
      ]
    }
  ]
}'

```



```
    }
  },
  {
    "type": "text",
    "text": "What are the key findings in this document?"
  }
]
}'
```

How PDF support works

When you send a PDF to Claude, these steps happen:

1. The system extracts the contents of the document:
 - Converts each page into an image.
 - Extracts the text from each page, provided alongside images.
 2. Claude analyzes both text and images to understand the document:
 - Supports visual elements like charts, diagrams, and non-textual content.
 3. Claude responds, referencing the PDF's contents if relevant:
 - Can reference both textual and visual content.
 - Use with prompt caching, batch processing, and tool use to optimize performance.
-

Estimate your costs

The token count of a PDF depends on:

- Text tokens: Typically 1,500-3,000 tokens per page depending on content density. Standard API pricing applies.
- Image tokens: Each page is converted into an image, with associated costs as per [image evaluation costs](#).

You can use [token counting](#) to estimate costs for your PDFs.

Optimize PDF processing

Improve performance

Follow these best practices:

- Place PDFs before text in requests
- Use standard fonts
- Ensure text is clear and legible
- Rotate pages to upright orientation
- Use logical page numbers (from PDF viewer)
- Split large PDFs into chunks
- Enable prompt caching for repeated analysis

Scale your implementation

For high-volume processing, consider:

Use prompt caching

Cache PDFs for repeated queries:

```
# Create a JSON request with cache control
jq -n --rawfile PDF_BASE64 pdf_base64.txt '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "document",
          "source": {
            "type": "base64",
            "media_type": "application/pdf",
            "data": $PDF_BASE64
          }
        },
        {
          "type": "text",
          "text": "Which model has the highest human preference win rates across each
        }
      ],
      "cache_control": {
        "type": "ephemeral"
      }
    }
  ]
}' > request.json

# Send the request
curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d @request.json
```

Process document batches

Use the [Message Batches API](#) for bulk workflows:

```
jq -n --rawfile PDF_BASE64 pdf_base64.txt '{
  "requests": [
    {
      "custom_id": "my-first-request",
      "params": {
        "model": "claude-opus-4-20250514",
        "max_tokens": 1024,
        "messages": [
          {
            "role": "user",
            "content": [
              {
                "type": "document",
                "source": {
```

```

        "type": "base64",
        "media_type": "application/pdf",
        "data": $PDF_BASE64
    }
},
{
    "type": "text",
    "text": "Extract 5 key insights from this document."
}
]
}
]
},
{
    "custom_id": "my-second-request",
    "params": {
        "model": "claude-opus-4-20250514",
        "max_tokens": 1024,
        "messages": [
            {
                "role": "user",
                "content": [
                    {
                        "type": "document",
                        "source": {
                            "type": "base64",
                            "media_type": "application/pdf",
                            "data": $PDF_BASE64
                        }
                    },
                    {
                        "type": "text",
                        "text": "Extract 5 key insights from this document."
                    }
                ]
            }
        ]
    }
}
]
}
]
}' > request_batches.json

```

Send batch request

```

curl https://api.anthropic.com/v1/messages/batches \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d @request_batches.json

```

Next steps

- [Try PDF examples](#): Explore practical examples of PDF processing.
- [View API reference](#): Complete API documentation for PDF support.

Was this page helpful?

[Yes](#) [No](#)

Links:

- [Vision](#)
- [Files API](#)

Follow us

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [Before you begin](#)
- [Check PDF requirements](#)
- [Supported platforms and models](#)
- [Process PDFs with Claude](#)
- [Send your first PDF request](#)
- [Option 1: URL-based PDF document](#)
- [Option 2: Base64-encoded PDF document](#)
- [Option 3: Files API](#)
- [How PDF support works](#)
- [Estimate your costs](#)
- [Optimize PDF processing](#)
 - [Improve performance](#)
 - [Scale your implementation](#)
 - [Use prompt caching](#)
 - [Process document batches](#)
- [Next steps](#)

Tool use with Claude - Anthropic

Overview

Claude is capable of interacting with tools and functions, allowing you to extend Claude's capabilities to perform a wider variety of tasks.

Learn everything you need to master tool use with Claude via our new comprehensive [tool use course](#)! Please continue to share your ideas and suggestions using this [form](#).

Example: Providing Tools to Claude via Messages API

Shell

```
curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "tools": [
      {
        "name": "get_weather",
        "description": "Get the current weather in a given location",
        "input_schema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"],
              "description": "The unit of temperature, either \"celsius\" or \"fahrenh"
            }
          },
          "required": ["location"]
        }
      }
    ],
    "messages": [
      {
        "role": "user",
        "content": "What is the weather like in San Francisco?"
      }
    ]
  }'
```

Claude will respond with a message similar to:

JSON

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-opus-4-20250514",
  "stop_reason": "tool_use",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "<thinking>I need to call the get_weather function, and the user wants S
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {
        "location": "San Francisco, CA",
        "unit": "celsius"
      }
    }
  ]
}
```

You then execute the `get_weather` function with the provided input, and return the result in a new `user` message:

Shell

```
curl https://api.anthropic.com/v1/messages \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "content-type: application/json" \
  -d '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "tools": [
      {
        "name": "get_weather",
        "description": "Get the current weather in a given location",
        "input_schema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"],
              "description": "The unit of temperature, either \"celsius\" or \"fahrenh"
            }
          }
        },
        "required": ["location"]
      }
    ],
    "messages": [
```

```
{
  "role": "user",
  "content": "What is the weather like in San Francisco?"
},
{
  "role": "tool_result",
  "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
  "content": "15 degrees"
}
]
}'
```

Claude then incorporates the weather data into its response:

JSON

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-opus-4-20250514",
  "stop_reason": "stop_sequence",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "The current weather in San Francisco is 15 degrees Celsius (59 degrees"
    }
  ]
}
```

Multiple Tool Usage

You can supply multiple tools for Claude to choose from in a single request, e.g., `get_weather` and `get_time`, to answer a complex query.

Example

```
curl https://api.anthropic.com/v1/messages \
-H "content-type: application/json" \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-d '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "tools": [
    {
      "name": "get_weather",
      "description": "Get the current weather in a given location",
      "input_schema": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "The city and state, e.g. San Francisco, CA"
          },
          "unit": {
            "type": "string",
```

```

        "enum": ["celsius", "fahrenheit"],
        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'
    }
},
"required": ["location"]
}
},
{
    "name": "get_time",
    "description": "Get the current time in a given time zone",
    "input_schema": {
        "type": "object",
        "properties": {
            "timezone": {
                "type": "string",
                "description": "The IANA time zone name, e.g., America/Los_Angeles"
            }
        }
    },
    "required": ["timezone"]
}
],
"messages": [
    {
        "role": "user",
        "content": "What is the weather like right now in New York? Also what time is
    }
]
}'

```

Claude will likely call `get_weather` first, then `get_time` based on the flow, executing each separately.

Handling Missing Information

If your prompt doesn't include enough info to fill all required parameters, Claude is likely to recognize the missing info and ask for it, especially when prompted to think before outputting a tool request. For example, asking "What's the weather?" without specifying a location may lead Claude to infer a default, e.g., `"New York, NY"`, but it might also ask for clarification.

Sequential Tools

Tasks requiring multiple tools in sequence, using one tool's output as the next input, are supported. Claude will call tools one at a time, optionally estimating parameters when dependent on previous tool results.

Example: Chain Location and Weather

Shell

```

curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,

```



```

"tools": [
  {
    "name": "get_location",
    "description": "Get the current user location based on their IP address. No pa
    "input_schema": {
      "type": "object",
      "properties": {}
    }
  },
  {
    "name": "get_weather",
    "description": "Get the current weather in a given location",
    "input_schema": {
      "type": "object",
      "properties": {
        "location": {
          "type": "string",
          "description": "The city and state, e.g., San Francisco, CA"
        },
        "unit": {
          "type": "string",
          "enum": ["celsius", "fahrenheit"],
          "description": "The unit of temperature"
        }
      },
      "required": ["location"]
    }
  }
],
"messages": [
  {
    "role": "user",
    "content": "What's the weather like where I am?"
  }
]
}'

```

Claude will call `get_location` first, then `get_weather` with the location.

Handling Incomplete Prompts

If the user prompt lacks details for a tool, Claude may ask for clarification or infer a default value.

Example Prompt: "What's the weather?"

Claude may respond with a parameter guess:

```

{
  "type": "tool_use",
  "id": "toolu_01A09q90qw90lq917835lq9",
  "name": "get_weather",
  "input": {
    "location": "New York, NY",
    "unit": "fahrenheit"
  }
}

```

Sequential Tool Calls & Dependencies

Claude can chain multiple tools, passing outputs from one as inputs to the next when appropriate, calling tools individually and returning results separately.

Chain of Thought Tool Use

By default, Claude Opus is prompted to think before calling a tool, analyzing in `<thinking>` tags. Claude Sonnet and Haiku are more aggressive, often calling tools even when not strictly necessary.

Use this prompt to encourage better assessment:

Prompt

Answer the user's request using relevant tools (if available). Before calling a tool,

JSON Mode Usage

Tools can produce JSON outputs matching schemas for structured data extraction or descriptions.

Example: Image Description with JSON Output

Shell

```
#!/bin/bash
IMAGE_URL="https://upload.wikimedia.org/wikipedia/commons/a/a7/Camponotus_flavomargina"
IMAGE_MEDIA_TYPE="image/jpeg"
IMAGE_BASE64=$(curl "$IMAGE_URL" | base64)

curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "tools": [
      {
        "name": "record_summary",
        "description": "Record summary of an image using well-structured JSON.",
        "input_schema": {
          "type": "object",
          "properties": {
            "key_colors": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "r": {"type": "number", "description": "red value [0.0, 1.0]"},
                  "g": {"type": "number", "description": "green value [0.0, 1.0]"},
                  "b": {"type": "number", "description": "blue value [0.0, 1.0]"},
                  "name": {"type": "string", "description": "Color name in snake_case"}
                }
              },
              "required": ["r", "g", "b", "name"]
            }
          }
        }
      }
    ]
  }
```

```

    },
    "description": "Key colors in the image. Limit to less than four."
  },
  "description": {
    "type": "string",
    "description": "Image description. Max of two sentences."
  },
  "estimated_year": {
    "type": "integer",
    "description": "Estimated year the image was taken, if a photograph."
  }
},
"required": ["key_colors", "description"]
}
],
"messages": [
  {
    "role": "user",
    "content": [
      {
        "type": "image",
        "source": {
          "type": "base64",
          "media_type": "'$IMAGE_MEDIA_TYPE'",
          "data": "'$IMAGE_BASE64'"
        }
      },
      {
        "type": "text",
        "text": "Describe this image."
      }
    ]
  }
]
}'

```

This enables structured description or analysis of images, even without executing tools.

Pricing

Tool use requests are priced based on:

1. The total input tokens sent to the model (including in the `tools` parameter)
2. The number of output tokens generated
3. Additional usage-based charges for server tools (e.g., web search costs)

Client-side tools are priced like standard API requests, while server-side tools may incur extra costs.

Additional token sources include:

- `<tools>` parameter in API requests (names, descriptions, schemas)
- `<tool_use>` content blocks in requests/responses

- `<tool_result>` content blocks in requests

When using tools, a special system prompt is automatically included for the model, with token counts outlined in the table below (excluding the additional tokens listed above).

Note: If no tools are provided, "none" as a tool choice uses 0 system prompt tokens.

Model	Tool choice	Token count
Claude Opus 4	auto / none / any / tool	346 / 313
Claude Sonnet 4	auto / none / any / tool	346 / 313
Claude Sonnet 3.7	auto / none / any / tool	346 / 313
Claude Sonnet 3.5 (Oct)	auto / none / any / tool	346 / 313
Claude Sonnet 3.5 (June)	auto / none / any / tool	294 / 261
Claude Haiku 3.5	auto / none / any / tool	264 / 340
Claude Opus 3	auto / none / any / tool	530 / 281
Claude Sonnet 3	auto / none / any / tool	159 / 235
Claude Haiku 3	auto / none / any / tool	264 / 340

These token counts are added to your input/output tokens to determine total request cost.

Next Steps

Explore our repository of ready-to-implement tool use code examples in our [cookbooks](#):

- [Calculator Tool](#): Learn integrating a simple calculator.
- [Customer Service Agent](#): Build a support bot.
- [JSON Extractor](#): Extract structured data.

Was this page helpful?

Yes No

[Google Sheets add-on](#)
[How to implement tool use](#)

Follow Us

- [X](#)
- [LinkedIn](#)
- [Discord](#)

On this page

- [How tool use works](#)
- [Client tools](#)
- [Server tools](#)
- [Tool use examples](#)
- [Pricing](#)
- [Next Steps](#)

How to implement tool use - Anthropic

Tools

How to implement tool use

Copy page

Choosing a model

Generally, use `Claude Opus 4`, `Claude Sonnet 4`, `Claude Sonnet 3.7`, `Claude Sonnet 3.5`, or `Claude Opus 3` for complex tools and ambiguous queries; they handle multiple tools better and seek clarification when needed.

Use `Claude Haiku 3.5` or `Claude Haiku 3` for straightforward tools, but note they may infer missing parameters.

If using Claude Sonnet 3.7 with tool use and extended thinking, refer to our guide [here](#) for more information.

Specifying client tools

Client tools (both Anthropic-defined and user-defined) are specified in the `tools` top-level parameter of the API request. Each tool definition includes:

Parameter	Description
<code>name</code>	The name of the tool. Must match the regex <code>^[a-zA-Z0-9_-]{1,64}\$</code> .
<code>description</code>	A detailed plaintext description of what the tool does, when it should be used, and how it behaves.
<code>input_schema</code>	A JSON Schema object defining the expected parameters for the tool.

Example simple tool definition

```
{
  "name": "get_weather",
  "description": "Get the current weather in a given location",
  "input_schema": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "The city and state, e.g. San Francisco, CA"
      },
      "unit": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "The unit of temperature, either 'celsius' or 'fahrenheit'"
      }
    }
  },
  "required": ["location"]
}
```

```
}  
}
```

This tool, named `get_weather`, expects an input object with a required `location` string and an optional `unit` string that must be either "celsius" or "fahrenheit".

Tool use system prompt

When you call the Anthropic API with the `tools` parameter, a special system prompt is constructed from the tool definitions, configuration, and any user prompt. It instructs the model on how to use the specified tools and provides context.

In this environment you have access to a set of tools you can use to answer the user's
{{ FORMATTING INSTRUCTIONS }}

String and scalar parameters should be specified as is, while lists and objects should
Here are the functions available in JSONSchema format:

```
{{ TOOL DEFINITIONS IN JSON SCHEMA }}  
{{ USER SYSTEM PROMPT }}  
{{ TOOL CONFIGURATION }}
```

Best practices for tool definitions

To maximize performance:

- **Provide extremely detailed descriptions.** Explain what the tool does, when to use it, what each parameter means, behavior, caveats, and limitations. Aim for 3-4 sentences per description.
- **Prioritize descriptions over examples.** Include examples only after a clear, comprehensive description.

Example of a good tool description

```
{  
  "name": "get_stock_price",  
  "description": "Retrieves the current stock price for a given ticker symbol. The tic  
  "input_schema": {  
    "type": "object",  
    "properties": {  
      "ticker": {  
        "type": "string",  
        "description": "The stock ticker symbol, e.g. AAPL for Apple Inc."  
      }  
    },  
    "required": ["ticker"]  
  }  
}
```

Poor example

```
{  
  "name": "get_stock_price",  
  "description": "Gets the stock price for a ticker.",  
  "input_schema": {  
    "type": "object",  
    "properties": {  
      "ticker": {  
        "type": "string"  
      }  
    }  
  }  
}
```

```

    },
    "required": ["ticker"]
  }
}

```

The good description clearly explains what the tool does, when to use it, what data it returns, and what the `ticker` parameter means.

Controlling Claude's output

Forcing tool use

You can specify the `tool_choice` parameter to dictate tool usage, such as:

```
{"type": "tool", "name": "get_weather"}
```

Options include:

- `auto`: Claude decides (default when tools are provided)
- `any`: Must use one of the provided tools, but not a specific one
- `tool`: Force usage of a specific tool
- `none`: Do not use tools (default when no tools)

Changing `tool_choice` when using prompt caching invalidates message blocks.

Additional notes

- When `tool_choice` is `any` or `tool`, the assistant is prefilled to enforce tool usage—no chain-of-thought content blocks will be emitted before tool outputs.
- For extended thinking, only `auto` and `none` are supported; `any` and `tool` are incompatible and will error.
- To retain chain-of-thought while using a specific tool, set `tool_choice` to `auto` and give explicit instructions in the user message.

JSON output

Tools can also be used to generate structured JSON responses by following a schema, e.g., with a `record_summary` tool.

Chain of thought

Claude often shows reasoning steps ("chain of thought") when using tools, especially in models like Opus 3, triggered by `tool_choice=auto`. The model may output reasoning in `<thinking>` tags, but format may change over time.

Example:

```

{
  "role": "assistant",
  "content": [
    {"type": "text", "text": "<thinking>Use get_weather for SF and get_time for LA</th
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA"}
    }
  ]
}

```

```
}
]
}
```

You can prompt Claude explicitly to show reasoning, e.g., "Explain your reasoning step-by-step in tags."

Parallel tool use

Claude may use multiple tools simultaneously, configurable via:

- `disable_parallel_tool_use=true`, which limits to one tool for `auto`, `any`, or `tool` modes.

For Claude Sonnet 3.7: enabling [token-efficient tool use](#) encourages parallel calling and reduces latency by ~14%. Alternatively, implement a "batch tool" that wraps multiple tools.

Handling tool use and tool result content blocks

Results from client tools

Responses include `stop_reason: "tool_use"` and content blocks:

- `id`: unique identifier
- `name`: tool name
- `input`: parameters passed to the tool

Example:

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-opus-4-20250514",
  "stop_reason": "tool_use",
  "role": "assistant",
  "content": [
    {"type": "text", "text": "<thinking>Decide to use get_weather for SF</thinking>"},
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": {"location": "San Francisco, CA", "unit": "celsius"}
    }
  ]
}
```

Processing the result

1. Extract `name`, `id`, and `input`.
2. Run the actual tool with `input`.
3. Continue conversation with a new user message containing:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
```



```

    "tool_use_id": "the same id",
    "content": "Tool output (string or nested content blocks)"
  }
]
}

```

4. If there is an error, include `is_error: true`.

Examples

Successful tool result

```

{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "15 degrees"
    }
  ]
}

```

Tool result with images

```

{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": [
        { "type": "text", "text": "15 degrees" },
        {
          "type": "image",
          "source": {
            "type": "base64",
            "media_type": "image/jpeg",
            "data": "/9j/4AAQSkZJRg..."
          }
        }
      ]
    }
  ]
}

```

Empty tool result

```

{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9"
    }
  ]
}

```

Handling results from server tools

Claude executes tools internally, and incorporates results directly. You do not need special handling for `is_error`.

Handling max tokens stop reason

If response is cut off at `max_tokens` and contains incomplete tool content, retry with higher `max_tokens`.

Handling pause_turn stop reason

For long-running tools like web search, if `pause_turn` occurs:

- Continue conversation by resubmitting the response content.
- Optionally modify content.
- Keep tool state.

Example in Python:

```
if response.stop_reason == "pause_turn":
    # Continue...
    messages = [
        {"role": "user", "content": "Search for..."},
        {"role": "assistant", "content": response.content}
    ]
    continuation = client.messages.create(model="claude-3-7-sonnet-latest", max_tokens
    print(continuation)
else:
    print(response)
```

Troubleshooting errors

Tool execution error

Return error message with `"is_error": true`. Example:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "ConnectionError: the weather service API is not available (HTTP 500)",
      "is_error": true
    }
  ]
}
```

Invalid tool name

If Claude cannot use a tool (missing required params), encode an error in `tool_result` and retry with more info.

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "Error: Missing required 'location' parameter",
      "is_error": true
    }
  ]
}
```

Server tool errors

For example, Web Search errors like rate limit exceeded (`too_many_requests`) or internal issues will be handled by Claude; no special handling needed.

Additional

- To prevent tags, add “Do not reflect on the quality of the returned search results in your response” to your prompt.
- For help on tools, see [overview](#) and [token-efficient use](#).

Token-efficient tool use - Anthropic

Overview

Claude Sonnet 3.7 is capable of calling tools in a token-efficient manner. Requests save an average of 14% in output tokens, up to 70%, which also reduces latency. Exact token reduction and latency improvements depend on the overall response shape and size.

Token-efficient tool use is a beta feature. Please make sure to evaluate your responses before using it in production.

Please use [this form](#) to provide feedback on the quality of the model responses, the API itself, or the quality of the documentation—we cannot wait to hear from you!

Recommendations

If you choose to experiment with this feature, we recommend using the **Prompt Improver** in the [Console](#) to improve your prompt.

Limitations

Token-efficient tool use does not currently work with [disabling parallel tool use](#).

Claude 4 models (Opus and Sonnet) do not support this feature. The beta header `token-efficient-tools-2025-02-19` will not break an API request, but it will result in a no-op.

How to use

To use this beta feature, add the header `token-efficient-tools-2025-02-19` to a tool use request. If you are using the SDK, ensure that you are using the beta SDK with `anthropic.beta.messages`.

Example API Request

```
curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "anthropic-beta: token-efficient-tools-2025-02-19" \
  -d '{
    "model": "claude-3-7-sonnet-20250219",
    "max_tokens": 1024,
    "tools": [
      {
        "name": "get_weather",
        "description": "Get the current weather in a given location",
        "input_schema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g. San Francisco, CA"
            }
          }
        }
      }
    ]
  }
```

```
      },
      "required": ["location"]
    }
  ],
  "messages": [
    {
      "role": "user",
      "content": "Tell me the weather in San Francisco."
    }
  ]
}' | jq '.usage'
```

The above request should, on average, use fewer input and output tokens than a normal request. To confirm this, try making the same request but remove `token-efficient-tools-2025-02-19` from the headers list.

Additional Notes

To keep the benefits of prompt caching, use the beta header consistently for requests you'd like to cache. If used selectively, prompt caching will fail.

Feedback

Was this page helpful?

[Yes](#) | [No](#)

Related

- [How to implement tool use](#)
 - [Fine-grained tool streaming](#)
-

Follow Us

[Twitter](#) | [LinkedIn](#) | [Discord](#)

Fine-grained tool streaming - Anthropic

Overview

Tool use now supports fine-grained streaming for parameter values. This allows developers to stream tool use parameters without buffering / JSON validation, reducing the latency to begin receiving large parameters.

Note: Fine-grained tool streaming is a beta feature. Please evaluate responses carefully before using in production.

Please use [this form](#) to provide feedback on model responses, the API, or documentation — we value your input!

Caution: When using fine-grained tool streaming, you may receive invalid or partial JSON inputs. Handle these edge cases in your code.

How to use fine-grained tool streaming

To activate this feature, add the header `fine-grained-tool-streaming-2025-05-14` to a tool use request and enable streaming.

Example API Usage

Shell

```
curl https://api.anthropic.com/v1/messages \
  -H "content-type: application/json" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "anthropic-beta: fine-grained-tool-streaming-2025-05-14" \
  -d '{
    "model": "claude-sonnet-4-20250514",
    "max_tokens": 65536,
    "tools": [
      {
        "name": "make_file",
        "description": "Write text to a file",
        "input_schema": {
          "type": "object",
          "properties": {
            "filename": {
              "type": "string",
              "description": "The filename to write text to"
            },
            "lines_of_text": {
              "type": "array",
              "description": "An array of lines of text to write to the file"
            }
          }
        },
        "required": ["filename", "lines_of_text"]
      }
    ]
  }
```

```

    }
  },
  "messages": [
    {
      "role": "user",
      "content": "Can you write a long poem and make a file called poem.txt?"
    }
  ],
  "stream": true
}' | jq '.usage'
```

Benefits of fine-grained streaming

In this example, Claude streams lines of a long poem into the `make_file` tool call without buffering to verify JSON validity. You can see parameter streams as they arrive, without waiting for full buffering.

Faster start, fewer word breaks

Tool chunks start streaming sooner, often longer, with fewer breaks — thanks to differences in chunking behavior.

Examples

Without fine-grained streaming (15s delay):

```

Chunk 1: '{" '
Chunk 2: 'query": "Ty'
Chunk 3: 'peScri'
Chunk 4: 'pt 5.0 5.1 '
Chunk 5: '5.2 5'
Chunk 6: '.3'
Chunk 8: ' new f'
Chunk 9: 'eatur'
...
```

With fine-grained streaming (3s delay):

```

Chunk 1: '{"query": "TypeScript 5.0 5.1 5.2 5.3'
Chunk 2: ' new features comparison'
```

Warning

Because parameters are sent without buffering or JSON validation, streams might be partial or invalid. If the `max_tokens` stop reason is reached, stream may end prematurely and be incomplete. Handle such cases in your implementation.

Feedback & Help

Was this page helpful?

Yes | No

Related links:

- [Token-efficient tool use](#)
- [Bash tool](#)

Follow us

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [How to use fine-grained tool streaming](#)

Bash tool

The bash tool enables Claude to execute shell commands in a persistent bash session, allowing system operations, script execution, and command-line automation.

Overview

The bash tool provides Claude with:

- Persistent bash session that maintains state
- Ability to run any shell command
- Access to environment variables and working directory
- Command chaining and scripting capabilities

Tool versions

Model	Tool Version
Claude 4 & Sonnet 3.7	bash_20250124
Claude Sonnet 3.5	bash_20241022

Claude Sonnet 3.5 requires the `computer-use-2024-10-22` beta header when using the bash tool. The bash tool is generally available in Claude 4 and Sonnet 3.7.

Use cases

- **Development workflows:** Run build commands, tests, and development tools
- **System automation:** Execute scripts, manage files, automate tasks
- **Data processing:** Process files, run analysis scripts, manage datasets
- **Environment setup:** Install packages, configure environments

Quick start

```
import anthropic

client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    tools=[
        {
            "type": "bash_20250124",
            "name": "bash"
        }
    ],
    messages=[
        {"role": "user", "content": "List all Python files in the current directory."}
    ]
)
```

How it works

The bash tool maintains a persistent session:

1. Claude determines what command to run
2. You execute the command in a bash shell
3. Return the output (stdout and stderr) to Claude
4. Session state persists between commands (environment variables, working directory)

Parameters

Parameter	Required	Description
<code>command</code>	Yes*	The bash command to run
<code>restart</code>	No	Set to <code>true</code> to restart the bash session

*Required unless using `restart`

Example usage

```
// Run a command
{
  "command": "ls -la *.py"
}
```

```
// Restart the session
{
  "restart": true
}
```

Example: Multi-step automation

Claude can chain commands to complete complex tasks:

```
# User request
"Install the requests library and create a simple Python script that fetches a joke fr

# Claude's tool uses:
# 1. Install package
{"command": "pip install requests"}

# 2. Create script
{"command": "cat > fetch_joke.py << 'EOF'\nimport requests\nresponse = requests.get('h

# 3. Run script
{"command": "python fetch_joke.py"}
```

The session maintains state between commands, so files created in step 2 are available in step 3.

Implement the bash tool

The bash tool is implemented as a schema-less tool. When using this tool, you don't need to provide an input schema as with other tools; the schema is built into Claude's model and can't be modified.

1. Set up a bash environment

```
import subprocess
import threading
import queue

class BashSession:
    def __init__(self):
        self.process = subprocess.Popen(
            ['/bin/bash'],
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            bufsize=0
        )
        self.output_queue = queue.Queue()
        self.error_queue = queue.Queue()
        self._start_readers()

    def _start_readers(self):
        def _read_output(pipe, queue):
            for line in iter(pipe.readline, ''):
                queue.put(line)
        threading.Thread(target=_read_output, args=(self.process.stdout, self.output_queue))
        threading.Thread(target=_read_output, args=(self.process.stderr, self.error_queue))
```

2. Handle command execution

```
def execute_command(self, command):
    # Send command to bash
    self.process.stdin.write(command + '\n')
    self.process.stdin.flush()
    # Capture output with timeout
    output = self._read_output(timeout=10)
    return output

def _read_output(self, timeout=10):
    output_lines = []
    try:
        while True:
            line = self.output_queue.get(timeout=timeout)
            output_lines.append(line)
    except queue.Empty:
        pass
    return ''.join(output_lines)
```

3. Process Claude's tool calls

Extract and execute commands from Claude's responses:

```
for content in response.content:
    if content.type == "tool_use" and content.name == "bash":
        if content.input.get("restart"):
            bash_session.restart()
```

```

        result = "Bash session restarted"
    else:
        command = content.input.get("command")
        result = bash_session.execute_command(command)
    # Return result to Claude
    tool_result = {
        "type": "tool_result",
        "tool_use_id": content.id,
        "content": result
    }

```

4. Implement safety measures

Add validation and restrictions:

```

def validate_command(command):
    # Block dangerous commands
    dangerous_patterns = ['rm -rf /', 'format', ':{:|:&};:']
    for pattern in dangerous_patterns:
        if pattern in command:
            return False, f"Command contains dangerous pattern: {pattern}"
    # Add more validation as needed
    return True, None

```

Handle errors

Command execution timeout

```

{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "Error: Command timed out after 30 seconds",
      "is_error": true
    }
  ]
}

```

Command not found

```

{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "bash: nonexistentcommand: command not found",
      "is_error": true
    }
  ]
}

```

Permission denied

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "bash: /root/sensitive-file: Permission denied",
      "is_error": true
    }
  ]
}
```

Follow implementation best practices

Use command timeouts

```
def execute_with_timeout(command, timeout=30):
    try:
        result = subprocess.run(
            command,
            shell=True,
            capture_output=True,
            text=True,
            timeout=timeout
        )
        return result.stdout + result.stderr
    except subprocess.TimeoutExpired:
        return f"Command timed out after {timeout} seconds"
```

Maintain session state

Commands run in the same session maintain state:

```
commands = [
    "cd /tmp",
    "echo 'Hello' > test.txt",
    "cat test.txt" # This works because we're still in /tmp
]
```

Handle large outputs

```
def truncate_output(output, max_lines=100):
    lines = output.split('\n')
    if len(lines) > max_lines:
        truncated = '\n'.join(lines[:max_lines])
        return f"{truncated}\n\n... Output truncated ({len(lines)} total lines) ..."
    return output
```

Log all commands

```
import logging

def log_command(command, output, user_id):
    logging.info(f"User {user_id} executed: {command}")
    logging.info(f"Output: {output[:200]}...") # Log first 200 chars
```

Sanitize outputs

```
def sanitize_output(output):
    import re
    # Remove potential secrets or credentials
    output = re.sub(r'aws_access_key_id\s*=\s*\S+', 'aws_access_key_id=***', output)
    output = re.sub(r'aws_secret_access_key\s*=\s*\S+', 'aws_secret_access_key=***', o
    return output
```

Security

The bash tool provides direct system access. Implement these safety measures:

- Running in isolated environments (Docker/VM)
- Implementing command filtering and allowlists
- Setting resource limits (CPU, memory, disk)
- Logging all executed commands

Key recommendations

- Use `ulimit` to set resource constraints
- Filter dangerous commands (`sudo` , `rm -rf` , etc.)
- Run with minimal user permissions
- Monitor and log all command execution

Pricing

The bash tool adds 245 input tokens to your API calls.

Additional tokens are consumed by:

- Command outputs (stdout/stderr)
- Error messages
- Large file contents

See [tool use pricing](#) for complete details.

Common patterns

Development workflows

- Running tests: `pytest --coverage --report`
- Building projects: `npm install --build`
- Git operations: `git status --git add . --git commit -m "message"`

File operations

- Processing data: `wc -l *.csv && ls -lh *.csv`
- Searching files: `find . -name "*.py" | xargs grep "pattern"`
- Creating backups: `tar -czf backup.tar.gz ./data`

System tasks

- Checking resources: `df -h && free -m`
- Process management: `ps aux | grep python`
- Environment setup: `export PATH=$PATH:/new/path && echo $PATH`

Limitations

- **No interactive commands:** Cannot handle `vim`, `less`, or password prompts
- **No GUI applications:** Command-line only
- **Session scope:** Persists within conversation, lost between API calls
- **Output limits:** Large outputs may be truncated
- **No streaming:** Results returned after completion

Combining with other tools

The bash tool is most powerful when combined with the [text editor](#) and other tools.

Next steps

- [Tool use overview](#)
- [Text editor tool](#)

Was this page helpful?

Yes | No

Follow-up links:

- [Fine-grained tool streaming](#)
- [Code execution tool](#)

Connect with us:

[Twitter](#) | [LinkedIn](#) | [Discord](#)

On this page

- [Overview](#)
- [Tool versions](#)
- [Use cases](#)
- [Quick start](#)
- [How it works](#)
- [Parameters](#)
- [Example: Multi-step automation](#)
- [Implement the bash tool](#)
- [Handle errors](#)
- [Follow implementation best practices](#)
- [Security](#)
- [Key recommendations](#)

- Pricing
- Common patterns
- Limitations
- Combining with other tools
- Next steps

Code execution tool - Anthropic

Tools

Overview

Code execution tool

The code execution tool allows Claude to execute Python code in a secure, sandboxed environment. Claude can analyze data, create visualizations, perform complex calculations, and process uploaded files directly within the API conversation.

The code execution tool is currently in beta.

This feature requires the [beta header](#) :

```
"anthropic-beta": "code-execution-2025-05-22"
```

Supported models

The code execution tool is available on:

- Claude Opus 4 (claude-opus-4-20250514)
- Claude Sonnet 4 (claude-sonnet-4-20250514)
- Claude Sonnet 3.7 (claude-3-7-sonnet-20250219)
- Claude Haiku 3.5 (claude-3-5-haiku-latest)

Quick start

Here's a simple example that asks Claude to perform a calculation:

```
curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "anthropic-beta: code-execution-2025-05-22" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 4096,
    "messages": [
      {
        "role": "user",
        "content": "Calculate the mean and standard deviation of [1, 2, 3, 4,
      ],
    ],
    "tools": [
      {
        "type": "code_execution_20250522",
        "name": "code_execution"
      }
    ]
  }
```

```
]
}'
```

How code execution works

When you add the code execution tool to your API request:

1. Claude evaluates whether code execution would help answer your question
2. Claude writes and executes Python code in a secure sandbox environment
3. Code execution may occur multiple times throughout a single request
4. Claude provides results with any generated charts, calculations, or analysis

Tool definition

The code execution tool requires no additional parameters:

```
{
  "type": "code_execution_20250522",
  "name": "code_execution"
}
```

Response format

Here's an example response with code execution:

```
{
  "role": "assistant",
  "container": {
    "id": "container_011CPR5CNjB747bTd36fQLFk",
    "expires_at": "2025-05-23T21:13:31.749448Z"
  },
  "content": [
    {
      "type": "text",
      "text": "I'll calculate the mean and standard deviation for you."
    },
    {
      "type": "server_tool_use",
      "id": "srvtoolu_01A2B3C4D5E6F7G8H9I0J1K2",
      "name": "code_execution",
      "input": {
        "code": "import numpy as np\nndata = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]\nmean = np"
      }
    },
    {
      "type": "code_execution_tool_result",
      "tool_use_id": "srvtoolu_01A2B3C4D5E6F7G8H9I0J1K2",
      "content": {
        "type": "code_execution_result",
        "stdout": "Mean: 5.5\nStandard deviation: 2.8722813232690143\n",
        "stderr": "",
        "return_code": 0
      }
    }
  ],
}
```

```

    {
      "type": "text",
      "text": "The mean of the dataset is 5.5 and the standard deviation is approximat
    }
  ],
  "id": "msg_01BqK2v4FnRs4xTjgL8EuZxz",
  "model": "claude-opus-4-20250514",
  "stop_reason": "end_turn",
  "usage": {
    "input_tokens": 45,
    "output_tokens": 187
  }
}

```

Results

Code execution results include:

- `stdout`: Output from print statements and successful execution
- `stderr`: Error messages if code execution fails
- `return_code` (0 for success, non-zero for failure)

Example:

```

{
  "type": "code_execution_tool_result",
  "tool_use_id": "srvtoolu_01ABC123",
  "content": {
    "type": "code_execution_result",
    "stdout": "",
    "stderr": "NameError: name 'undefined_variable' is not defined",
    "return_code": 1
  }
}

```

Errors

If there is an error using the tool there will be a `code_execution_tool_result_error`:

```

{
  "type": "code_execution_tool_result_error",
  "error_code": "unavailable"
}

```

Possible errors include:

- `unavailable`: The code execution tool is unavailable
- `code_execution_exceeded`: Execution time exceeded the maximum allowed
- `container_expired`: The container is expired and not available

pause_turn stop reason

The response may include a `pause_turn` stop reason, which indicates that the API paused a long-running turn. You may provide the response back as-is in a subsequent request to let Claude continue its turn, or modify the content if you wish to interrupt the conversation.

Working with Files in Code Execution

Code execution can analyze files uploaded via the Files API, such as CSV files, Excel files, and other data formats. This allows Claude to read, process, and generate insights from your data. You can pass multiple files per request.

Using the Files API with Code Execution requires two beta headers:

```
"anthropic-beta": "code-execution-2025-05-22,files-api-2025-04-14"
```

Supported file types

The Python environment is capable of working with but not limited to the following file types:

- CSV
- Excel (`.xlsx` , `.xls`)
- JSON
- XML
- Images (JPEG, PNG, GIF, WebP)
- Text files (`.txt` , `.md` , `.py` , etc)

Loading files for code execution

1. Upload your file using the [Files API](#)
2. Reference the file in your message using a `container_upload` content block
3. Include the code execution tool in your API request

First, upload a file

```
curl https://api.anthropic.com/v1/files \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "anthropic-beta: files-api-2025-04-14" \
  --form 'file=@"data.csv"'
```

Then use the file_id with code execution

```
curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "anthropic-beta: code-execution-2025-05-22,files-api-2025-04-14" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 4096,
    "messages": [
      {
        "role": "user",
        "content": [
          {"type": "text", "text": "Analyze this CSV data"},
          {"type": "container_upload", "file_id": "file_abc123"}
        ]
      }
    ],
    "tools": [
      {
        "type": "code_execution_20250522",
        "name": "code_execution"
      }
    ]
  }'
```

```

    }
  ]
}'

```

Retrieving files created by code execution

When Claude creates files during code execution (e.g., saving matplotlib plots, generating CSVs), you can retrieve these files using the Files API:

```

from anthropic import Anthropic

# Initialize the client
client = Anthropic()

# Request code execution that creates files
response = client.beta.messages.create(
    model="claude-opus-4-20250514",
    betas=["code-execution-2025-05-22", "files-api-2025-04-14"],
    max_tokens=4096,
    messages=[
        {"role": "user", "content": "Create a matplotlib visualization and save it as"},
    ],
    tools=[
        {"type": "code_execution_20250522", "name": "code_execution"}
    ]
)

# Extract file IDs from the response
def extract_file_ids(response):
    file_ids = []
    for item in response.content:
        if item.type == "code_execution_tool_result":
            content_item = item.content
            if content_item.get('type') == 'code_execution_result':
                for file in content_item.get('content', []):
                    file_ids.append(file['file_id'])
    return file_ids

# Download the created files
for file_id in extract_file_ids(response):
    file_metadata = client.beta.files.retrieve_metadata(file_id)
    file_content = client.beta.files.download(file_id)
    file_content.write_to_file(file_metadata.filename)
    print(f"Downloaded: {file_metadata.filename}")

```

Containers

The code execution tool runs in a secure, containerized environment designed specifically for Python code execution.

Runtime environment

- Python version: 3.11.12
- Operating system: Linux-based container
- Architecture: x86_64 (AMD64)

Resource limits

- Memory: 1GiB RAM
- Disk space: 5GiB workspace storage
- CPU: 1 CPU

Networking and security

- Internet access: Disabled for security
- External connections: No outbound network requests permitted
- Sandbox isolation: Full isolation from host system and other containers
- File access: Limited to workspace directory only
- Workspace scoping: Containers scoped to the API key workspace
- Expiration: Containers expire 1 hour after creation

Pre-installed libraries

The sandboxed Python environment includes:

- Data Science: pandas, numpy, scipy, scikit-learn, statsmodels
- Visualization: matplotlib
- File Processing: pyarrow, openpyxl, xlrd, pillow
- Math & Computing: sympy, mpmath
- Utilities: tqdm, python-dateutil, pytz, joblib

Container reuse

You can reuse an existing container across multiple API requests by providing the container ID from a previous response. This maintains created files between requests.

Example

```
import os
from anthropic import Anthropic

# Initialize the client
client = Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))

# First request: Create a file with a random number
response1 = client.beta.messages.create(
    model="claude-opus-4-20250514",
    betas=["code-execution-2025-05-22"],
    max_tokens=4096,
    messages=[
        {"role": "user", "content": "Write a file with a random number and save it to"},
    ],
    tools=[
        {"type": "code_execution_20250522", "name": "code_execution"}
    ]
)

# Get container ID
container_id = response1.container.id

# Second request: Reuse the container to read the file
```

```

response2 = client.beta.messages.create(
    container=container_id,
    model="claude-opus-4-20250514",
    betas=["code-execution-2025-05-22"],
    max_tokens=4096,
    messages=[
        {"role": "user", "content": "Read the number from '/tmp/number.txt' and calcul
    ],
    tools=[
        {"type": "code_execution_20250522", "name": "code_execution"}
    ]
)

```

Example

Reusing container to process files

```

import os
from anthropic import Anthropic

# Initialize the client
client = Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))

# First request: Create a file with a random number
response1 = client.beta.messages.create(
    model="claude-opus-4-20250514",
    betas=["code-execution-2025-05-22"],
    max_tokens=4096,
    messages=[
        {"role": "user", "content": "Write a file with a random number and save it to
    ],
    tools=[
        {"type": "code_execution_20250522", "name": "code_execution"}
    ]
)

# Save container ID
container_id = response1.container.id

# Second request: Use container to read and process the file
response2 = client.beta.messages.create(
    container=container_id,
    model="claude-opus-4-20250514",
    betas=["code-execution-2025-05-22"],
    max_tokens=4096,
    messages=[
        {"role": "user", "content": "Read the number from '/tmp/number.txt' and calcul
    ],
    tools=[
        {"type": "code_execution_20250522", "name": "code_execution"}
    ]
)

```

Streaming

With streaming enabled, you'll receive code execution events as they occur:

(Examples of stream events omitted for brevity)

Batch requests

You can include the code execution tool in the [Messages Batches API](#). Code execution requests through the batch API are priced the same as in regular Messages API requests.

Usage and pricing

The code execution tool usage is tracked separately from token usage.

Execution time is a minimum of 5 minutes.

If files are included in the request, execution time is billed even if the tool is not used due to files being preloaded onto the container.

Pricing: \$0.05 per session-hour.

Was this page helpful?

Yes | No

[bash](#) | [python](#) | [shell](#)

[Links to tools](#) | [Community](#) | [Legal & Privacy](#)

Overview

Claude can interact with computer environments through the computer use tool, which provides screenshot capabilities and mouse/keyboard control for autonomous desktop interaction.

Model compatibility

Model	Tool Version	Beta Flag
Claude 4 Opus & Sonnet	computer_20250124	computer-use-2025-01-24
Claude Sonnet 3.7	computer_20250124	computer-use-2025-01-24
Claude Sonnet 3.5 (new)	computer_20241022	computer-use-2024-10-22

Claude 4 models use updated tool versions optimized for the new architecture. Claude Sonnet 3.7 introduces additional capabilities including the thinking feature for more insight into the model's reasoning process.

Security considerations

Computer use is a beta feature with unique risks. Precautions include:

1. Use a dedicated virtual machine or container with minimal privileges.
2. Avoid giving access to sensitive data.
3. Limit internet access to an allowlist of domains.
4. Obtain user confirmation for actions with real-world consequences.

Claude will may follow commands in webpage content or images that could override instructions. We offer classifiers to flag prompt injections, which can prompt Claude for confirmation before proceeding. To disable this, [contact us](#).

Always review and verify actions and logs manually.

Reference implementation

[Get started quickly with our computer use reference implementation](#) including web interface, Docker container, example tools, and agent loop.

How computer use works

1. Provide Claude with the computer use tool and a user prompt requiring desktop interaction.
2. Claude assesses whether to use the tool, constructs a request, and signals intent with `stop_reason` of `tool_use`.
3. Your system executes the tool, then sends results back to Claude.
4. Claude continues calling tools until task completion or max iteration limit.

This loop is called the “agent loop.”

The computing environment

Includes:

- Virtual display via X11 (Xvfb)

- Desktop environment with window manager (Mutter)
- Pre-installed applications (Firefox, LibreOffice, etc.)
- Tool implementations translating Claude's requests
- An agent loop managing communication

Your application relays Claude's requests to this environment and returns results, running inside a Docker container for security.

How to implement computer use

Start with our reference implementation

- [GitHub repo](#)
- Includes Docker setup, tool implementations, agent loop, and web UI.

Understanding the multi-agent loop

The core cycle:

- Claude requests actions
- Your system executes them
- Results are sent back
- Loop continues until task completion.

Example pseudo-code is demonstrated in the documentation.

Handle errors

Examples include:

- Screenshot failures
- Invalid coordinates
- Action execution errors

Implement appropriate error messages, validation, and logging.

Best practices

- Use display resolutions like 1024x768 or 1280x800.
- Encode screenshots as base64 PNG/JPEG.
- Include delays and verify actions before executing.
- Log actions for debugging.

Limitations

- Latency may be slower than human actions.
- Vision and tool selection may be imperfect.
- Scrolling and spreadsheet interactions may have reliability issues.
- Risks of prompt injection, prompt hacking, and content overrides exist.
- Use within trusted environments and verify outputs carefully.

Pricing

- Adds approximately 466-499 tokens to system prompt overhead.
- Tokens per tool usage vary by model (e.g., 735 for Claude 4 / Sonnet 3.7).

- Additional costs for images and tool results.
- Review [pricing details](#).

Next steps

- [Docker-based reference implementation](#)
- [Tool documentation](#)

Feedback?

[Provide feedback here](#).

Follow us on [X](#), [LinkedIn](#), [Discord](#).

Text editor tool - Anthropic

Overview

Claude can use an Anthropic-defined text editor tool to view and modify text files, helping you debug, fix, and improve your code or other text documents. This allows Claude to directly interact with your files, providing hands-on assistance rather than just suggesting changes.

Before using the text editor tool

Use a compatible model

Anthropic's text editor tool is available for multiple Claude models:

- **Claude 4 Opus & Sonnet:** `text_editor_20250429`
- **Claude Sonnet 3.7:** `text_editor_20250124`
- **Claude Sonnet 3.5:** `text_editor_20241022`

It is generally available in Claude 4 and Sonnet 3.7. The newer `text_editor_20250429` for Claude 4 models does not include the `undo_edit` command. For undo functionality, use Claude 3.7 or Sonnet 3.5.

Assess your use case fit

Some examples of when to use the text editor tool are:

- **Code debugging:** Identify and fix bugs
- **Code refactoring:** Improve code structure and readability
- **Documentation generation:** Add docstrings, comments, or README files
- **Test creation:** Generate unit tests based on implementation

Use the text editor tool

Provide the tool to Claude via the Messages API with the name `str_replace_based_edit_tool`.

API example request

```
curl https://api.anthropic.com/v1/messages \
-H "content-type: application/json" \
-H "x-api-key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-d '{
  "model": "claude-opus-4-20250514",
  "max_tokens": 1024,
  "tools": [
    {
      "type": "text_editor_20250429",
      "name": "str_replace_based_edit_tool"
    }
  ],
  "messages": [
    {
```

```

        "role": "user",
        "content": "There\'s a syntax error in my primes.py file. Can you help me fix
    }
]
}'

```

Command reference

view

Examines file contents or lists directory:

- **Parameters:**
 - `command`: must be `view`
 - `path`: to file/directory
 - `view_range` (optional): `[start_line, end_line]`, for partial views.

Example:

```

{
  "type": "tool_use",
  "command": "view",
  "path": "primes.py"
}

```

or for directory:

```

{
  "type": "tool_use",
  "command": "view",
  "path": "src/"
}

```

str_replace

Replaces a specific string in a file:

- **Parameters:**
 - `command`: `str_replace`
 - `path`: target file
 - `old_str`: string to replace (exact match)
 - `new_str`: new string

Example:

```

{
  "type": "tool_use",
  "command": "str_replace",
  "path": "primes.py",
  "old_str": "for num in range(2, limit + 1)",
  "new_str": "for num in range(2, limit + 1):"
}

```

create

Creates a new file with specified content:

- **Parameters:**
 - `command` : create
 - `path` : file path
 - `file_text` : content

Example:

```
{
  "type": "tool_use",
  "command": "create",
  "path": "test_primes.py",
  "file_text": "import unittest\nimport primes\n\nclass TestPrimes(unittest.TestCase):
}
```

insert

Inserts text at a specific line:

- **Parameters:**
 - `command` : insert
 - `path` : file path
 - `insert_line` : line number (0-based)
 - `new_str` : text to insert

Example:

```
{
  "type": "tool_use",
  "command": "insert",
  "path": "primes.py",
  "insert_line": 0,
  "new_str": "\n\nModule for working with prime numbers.\n\nThis module provides fun
}
```

undo_edit

Reverts last edit:

- **Note:** Only available for Claude Sonnet 3.7 and 3.5, not in Claude 4 with `text_editor_20250429`.
- **Parameters:**
 - `command` : undo_edit
 - `path` : target file

Usage flow

1. Claude uses the tool commands (`view`, `str_replace`, etc.) to examine and modify files.
2. Before making edits, it's recommended to backup files.
3. For replacements, ensure exact matching and only one match unless intentional.
4. After changes, verify by running tests or checks.
5. Handle errors with proper error messages, e.g., file not found, permission denied, multiple/no matches.

Implementing the tool

1. Initialize file operation helpers.
2. Process Claude's commands with validation and security:

- Path validation
 - Backup creation
 - Error handling
3. Parse Claude's responses to identify and execute tool commands.
 4. Return results to Claude, including success info or errors.

Error handling

- **File not found:**

```
{"role": "user", "content": [{"type": "tool_result", "tool_use_id": "...", "content":
```

- **Multiple matches in `str_replace` :**

```
{"role": "user", "content": [{"type": "tool_result", "tool_use_id": "...", "content":
```

- **No match for replacement:**

```
{"role": "user", "content": [{"type": "tool_result", "tool_use_id": "...", "content":
```

- **Permission issues:**

```
{"role": "user", "content": [{"type": "tool_result", "tool_use_id": "...", "content":
```

Best practices

- Be specific and clear in your prompts.
- Validate file paths.
- Backup files before modifications.
- Ensure precise matching for replacements.
- After edits, verify functionality.

Pricing and tokens

The text editor tool follows the same token pricing as other Claude tools. Additional input tokens needed:

Tool	Additional input tokens
text_editor_20250429 (Claude 4)	700 tokens
text_editor_20250124 (Claude Sonnet 3.7)	700 tokens
text_editor_20241022 (Claude Sonnet 3.5)	700 tokens

For detailed pricing, see [Tool use pricing](#).

Next steps

- Integrate into your development workflows.
- Automate documentation and testing.
- Build custom assistants for debugging and code review.
- Convert file formats automatically.

Change log

Date	Version	Changes
April 29, 2025	text_editor_20250429	Release for Claude 4; removes <code>undo_edit</code> command
March 13, 2025	text_editor_20250124	Documentation; optimized for Claude Sonnet 3.7

Date	Version	Changes
October 22, 2024	<code>text_editor_20241022</code>	Initial release for Claude Sonnet 3.5

On this page

- [Before using the text editor tool](#)
- [Use a compatible model](#)
- [Assess your use case fit](#)
- [Use the text editor tool](#)
- [Text editor commands](#)
- [view](#)
- [str_replace](#)
- [create](#)
- [insert](#)
- [undo_edit](#)
- [Example: Fixing a syntax error](#)
- [Implementing the tool](#)
- [Handle errors](#)
- [Best practices](#)
- [Pricing & Token Usage](#)
- [Integrate with other tools](#)
- [Change log](#)
- [Next steps](#)

Web search tool - Anthropic

Tools

Web search tool

The web search tool gives Claude direct access to real-time web content, allowing it to answer questions with up-to-date information beyond its knowledge cutoff. Claude automatically cites sources from search results as part of its answer.

Please reach out through our [feedback form](#) to share your experience with the web search tool.

Supported models

Web search is available on:

- Claude Opus 4 (claude-opus-4-20250514)
- Claude Sonnet 4 (claude-sonnet-4-20250514)
- Claude Sonnet 3.7 (claude-3-7-sonnet-20250219)
- Claude Sonnet 3.5 (new) (claude-3-5-sonnet-latest)
- Claude Haiku 3.5 (claude-3-5-haiku-latest)

How web search works

When you add the web search tool to your API request:

1. Claude decides when to search based on the prompt.
2. The API executes the searches and provides Claude with the results. This process may repeat multiple times throughout a single request.
3. At the end of its turn, Claude provides a final response with cited sources.

How to use web search

Your organization's administrator must enable web search in [Console](#).

Provide the web search tool in your API request:

```
curl https://api.anthropic.com/v1/messages \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --data '{
    "model": "claude-opus-4-20250514",
    "max_tokens": 1024,
    "messages": [
      {
        "role": "user",
        "content": "How do I update a web app to TypeScript 5.5?"
      }
    ],
    "tools": [
```

```

    {
      "type": "web_search_20250305",
      "name": "web_search",
      "max_uses": 5
    }
  ]
}'

```

Tool definition

The web search tool supports the following parameters:

```

{
  "type": "web_search_20250305",
  "name": "web_search",
  // Optional: Limit the number of searches per request
  "max_uses": 5,

  // Optional: Only include results from these domains
  "allowed_domains": ["example.com", "trusteddomain.org"],

  // Optional: Never include results from these domains
  "blocked_domains": ["untrustedsource.com"],

  // Optional: Localize search results
  "user_location": {
    "type": "approximate",
    "city": "San Francisco",
    "region": "California",
    "country": "US",
    "timezone": "America/Los_Angeles"
  }
}

```

Max uses

The `max_uses` parameter limits the number of searches performed. If Claude attempts more searches than allowed, the `web_search_tool_result` will be an error with the `max_uses_exceeded` error code.

Domain filtering

When using domain filters:

- Domains should not include the HTTP/HTTPS scheme (use `example.com` instead of `https://example.com`)
- Subdomains are automatically included (`example.com` covers `docs.example.com`)
- Subpaths are supported (`example.com/blog`)
- You can use either `allowed_domains` or `blocked_domains`, but not both in the same request.

Localization

The `user_location` parameter allows you to localize search results based on a user's location:

- `type`: must be `approximate`
- `city`: the city name

- `region` : the region or state
- `country` : the country
- `timezone` : the [IANA timezone ID](#)

Response

Here's an example response structure:

```
{
  "role": "assistant",
  "content": [
    // 1. Claude's decision to search
    {
      "type": "text",
      "text": "I'll search for when Claude Shannon was born."
    },
    // 2. The search query used
    {
      "type": "server_tool_use",
      "id": "srvtoolu_01WYG3ziw53XMcoyKL4XcZmE",
      "name": "web_search",
      "input": {
        "query": "claudio shannon birth date"
      }
    },
    // 3. Search results
    {
      "type": "web_search_tool_result",
      "tool_use_id": "srvtoolu_01WYG3ziw53XMcoyKL4XcZmE",
      "content": [
        {
          "type": "web_search_result",
          "url": "https://en.wikipedia.org/wiki/Claude_Shannon",
          "title": "Claude Shannon - Wikipedia",
          "encrypted_content": "EqgfCioIARgBIiQ3YTAwMjY1MmZjM5LTQ1NGUtODgxNC1kNjNjNT",
          "page_age": "April 30, 2025"
        }
      ]
    },
    {
      "text": "Based on the search results, ",
      "type": "text"
    },
    // 4. Claude's response with citations
    {
      "text": "Claude Shannon was born on April 30, 1916, in Petoskey, Michigan",
      "type": "text",
      "citations": [
        {
          "type": "web_search_result_location",
          "url": "https://en.wikipedia.org/wiki/Claude_Shannon",
          "title": "Claude Shannon - Wikipedia",
          "encrypted_index": "Eo8BCioIAhgBIiQyYjQ00WJmZi1lNm..",
          "cited_text": "Claude Elwood Shannon (April 30, 1916 - February 24, 2001) wa"
        }
      ]
    }
  ]
}
```

```

    ]
  }
],
"id": "msg_a930390d3a",
"usage": {
  "input_tokens": 6039,
  "output_tokens": 931,
  "server_tool_use": {
    "web_search_requests": 1
  }
},
"stop_reason": "end_turn"
}

```

Search results

Search results include:

- `url` : The URL of the source page
- `title` : The title of the source page
- `page_age` : When the site was last updated
- `encrypted_content` : Encrypted content that must be passed back in multi-turn conversations for citations

Citations

Citations are always enabled for web search, and each `web_search_result_location` includes:

- `url` : The URL of the cited source
- `title` : The title of the cited source
- `encrypted_index` : A reference that must be passed back for multi-turn conversations
- `cited_text` : Up to 150 characters of the cited content

The web search citation fields `cited_text`, `title`, and `url` do not count towards input or output token usage.

When displaying web results or information contained in web results to end users, inline citations must be made clearly visible and clickable in your user interface.

Errors

If an error occurs during web search, you'll receive a response like:

```

{
  "type": "web_search_tool_result",
  "tool_use_id": "servertoolu_a93jad",
  "content": {
    "type": "web_search_tool_result_error",
    "error_code": "max_uses_exceeded"
  }
}

```

Possible error codes:

- `too_many_requests` : Rate limit exceeded

- `invalid_input` : Invalid search query parameter
- `max_uses_exceeded` : Maximum web search tool uses exceeded
- `query_too_long` : Query exceeds maximum length
- `unavailable` : An internal error occurred

pause_turn stop reason

The response may include a `pause_turn` stop reason, indicating that the API paused a long-running turn. You may provide the response back as-is in a subsequent request to let Claude continue its turn, or modify the content to interrupt the conversation.

Prompt caching

Web search works with [prompt caching](#). To enable prompt caching, add at least one `cache_control` breakpoint in your request. The system will automatically cache until the last `web_search_tool_result` block during execution.

For multi-turn conversations, set a `cache_control` breakpoint after the last `web_search_tool_result` to reuse cached content.

Example of using prompt caching:

```
import anthropic

client = anthropic.Anthropic()

# First request with web search and cache breakpoint
messages = [
    {
        "role": "user",
        "content": "What's the current weather in San Francisco today?"
    }
]

response1 = client.messages.create(
    model="claude-opus-4-20250514",
    max_tokens=1024,
    messages=messages,
    tools=[
        {
            "type": "web_search_20250305",
            "name": "web_search",
            "user_location": {
                "type": "approximate",
                "city": "San Francisco",
                "region": "California",
                "country": "US",
                "timezone": "America/Los_Angeles"
            }
        }
    ]
)

# Add Claude's response to the conversation
messages.append({"role": "assistant", "content": response1.content})
```

```

# Second request with cache breakpoint after the search results
messages.append({
    "role": "user",
    "content": "Should I expect rain later this week?",
    "cache_control": {"type": "ephemeral"} # Cache up to this point
})

response2 = client.messages.create(
    model="claude-opus-4-20250514",
    max_tokens=1024,
    messages=messages,
    tools=[
        {
            "type": "web_search_20250305",
            "name": "web_search",
            "user_location": {
                "type": "approximate",
                "city": "San Francisco",
                "region": "California",
                "country": "US",
                "timezone": "America/Los_Angeles"
            }
        }
    ]
)

# The second response benefits from cached search results
print(f"Cache read tokens: {response2.usage.get('cache_read_input_tokens', 0)}")

```

Streaming

With streaming enabled, you'll receive search events as part of the stream. There will be a pause while the search executes:

```

event: message_start
data: {"type": "message_start", "message": {"id": "msg_abc123", "type": "message"}}

event: content_block_start
data: {"type": "content_block_start", "index": 0, "content_block": {"type": "text", "t

# Claude's decision to search

event: content_block_start
data: {"type": "content_block_start", "index": 1, "content_block": {"type": "server_to

# Search query streamed
event: content_block_delta
data: {"type": "content_block_delta", "index": 1, "delta": {"type": "input_json_delta"

# Pause while search executes

# Search results streamed
event: content_block_start
data: {"type": "content_block_start", "index": 2, "content_block": {"type": "web_searc

```

Claude's response with citations (omitted in this example)

Batch requests

You can include the web search tool in the [Messages Batches API](#). Calls through this API are priced the same as regular message requests.

Usage and pricing

Web search usage is charged in addition to token usage:

```
{
  "usage": {
    "input_tokens": 105,
    "output_tokens": 6039,
    "cache_read_input_tokens": 7123,
    "cache_creation_input_tokens": 7345,
    "server_tool_use": {
      "web_search_requests": 1
    }
  }
}
```

Web search costs **\$10 per 1,000 searches**, plus standard token costs for generated content. Results retrieved during a conversation count as input tokens across multiple turns. Each search counts as one use, regardless of results returned. If an error occurs, the search will not be billed.

Was this page helpful?

[Yes](#) / [No](#)

[Text editor tool](#) | [MCP connector](#)

On this page

- [Supported models](#)
- [How web search works](#)
- [How to use web search](#)
- [Tool definition](#)
- [Max uses](#)
- [Domain filtering](#)
- [Localization](#)
- [Response](#)
- [Search results](#)
- [Citations](#)
- [Errors](#)
- [pause_turn stop reason](#)
- [Prompt caching](#)
- [Streaming](#)
- [Batch requests](#)
- [Usage and pricing](#)

MCP connector - Anthropic

Introduction

Claude's Model Context Protocol (MCP) connector enables connecting to remote MCP servers directly from the Messages API without needing a separate MCP client.

This feature requires the beta header: `"anthropic-beta": "mcp-client-2025-04-04"`.

Key features

- **Direct API integration:** Connect to MCP servers without implementing an MCP client
- **Tool calling support:** Access MCP tools through the Messages API
- **OAuth authentication:** Support for OAuth Bearer tokens for authenticated servers
- **Multiple servers:** Connect to multiple MCP servers in a single request

Limitations

- Only `MCP specification`, specifically `tool calls`, are supported.
- The server must be publicly accessible via HTTP. Local STDIO servers are not supported.
- Not supported on Amazon Bedrock and Google Vertex.

Using the MCP connector in the Messages API

To connect to a remote MCP server, include the `mcp_servers` parameter in your API request:

```
curl https://api.anthropic.com/v1/messages \
-H "Content-Type: application/json" \
-H "X-API-Key: $ANTHROPIC_API_KEY" \
-H "anthropic-version: 2023-06-01" \
-H "anthropic-beta: mcp-client-2025-04-04" \
-d '{
  "model": "claude-sonnet-4-20250514",
  "max_tokens": 1000,
  "messages": [{"role": "user", "content": "What tools do you have available?"}],
  "mcp_servers": [
    {
      "type": "url",
      "url": "https://example-server.modelcontextprotocol.io/sse",
      "name": "example-mcp",
      "authorization_token": "YOUR_TOKEN"
    }
  ]
}'
```

MCP server configuration

Each MCP server in the `mcp_servers` array supports the following configuration:

```
{
  "type": "url",
  "url": "https://example-server.modelcontextprotocol.io/sse",
  "name": "example-mcp",
  "tool_configuration": {
    "enabled": true,
    "allowed_tools": ["example_tool_1", "example_tool_2"]
  },
  "authorization_token": "YOUR_TOKEN"
}
```

Field descriptions

Property	Type	Required	Description
<code>type</code>	string	Yes	Currently only "url" is supported
<code>url</code>	string	Yes	The URL of the MCP server. Must start with <code>https://</code>
<code>name</code>	string	Yes	Unique identifier for the MCP server, used to disambiguate tools
<code>tool_configuration</code>	object	No	Configure tool usage
<code>tool_configuration.enabled</code>	boolean	No	Enable tools from this server (default: true)
<code>tool_configuration.allowed_tools</code>	array	No	List of allowed tools; all tools are allowed by default
<code>authorization_token</code>	string	No	OAuth token if required by the MCP server

Response content types

When Claude uses MCP tools, responses include new block types:

MCP Tool Use Block

```
{
  "type": "mcp_tool_use",
  "id": "mcp_toolu_014Q35RayjACSWkSj4X2yov1",
  "name": "echo",
  "server_name": "example-mcp",
  "input": {
    "param1": "value1",
    "param2": "value2"
  }
}
```

MCP Tool Result Block

```
{
  "type": "mcp_tool_result",
  "tool_use_id": "mcp_toolu_014Q35RayjACSWkSj4X2yov1",
  "is_error": false,
  "content": [
    {
      "type": "text",

```

```
    "text": "Hello"
  }
]
}
```

Multiple MCP servers

You can connect to multiple MCP servers by including multiple objects in the `mcp_servers` array:

```
{
  "model": "claude-sonnet-4-20250514",
  "max_tokens": 1000,
  "messages": [
    {
      "role": "user",
      "content": "Use tools from both mcp-server-1 and mcp-server-2 to complete this t"
    }
  ],
  "mcp_servers": [
    {
      "type": "url",
      "url": "https://mcp.example1.com/sse",
      "name": "mcp-server-1",
      "authorization_token": "TOKEN1"
    },
    {
      "type": "url",
      "url": "https://mcp.example2.com/sse",
      "name": "mcp-server-2",
      "authorization_token": "TOKEN2"
    }
  ]
}
```

Authentication

MCP servers requiring OAuth need an `authorization_token`. The OAuth flow can be guided with the MCP inspector:

1. Run: `npx @modelcontextprotocol/inspector` (requires Node.js)
2. Select transport type (`SSE` or `Streamable HTTP`)
3. Enter MCP server URL
4. Click “Open Auth Settings”
5. Use “Quick OAuth Flow” to authorize
6. Follow the steps; copy `access_token`
7. Paste the token into `authorization_token` in server config

Using the access token

Use it in your server config as:

```
{
  "mcp_servers": [
    {
```

```
    "type": "url",
    "url": "https://example-server.modelcontextprotocol.io/sse",
    "name": "authenticated-server",
    "authorization_token": "YOUR_ACCESS_TOKEN_HERE"
  }
]
}
```

For detailed OAuth explanations, see the [MCP specification](#).

Was this page helpful?

[Yes](#) | [No](#)

Links and social

- [Web search tool](#)
- [Remote MCP servers](#)

  

On this page

- [Key features](#)
- [Limitations](#)
- [Using the MCP connector in the Messages API](#)
- [MCP server configuration](#)
- [Field descriptions](#)
- [Response content types](#)
- [MCP Tool Use Block](#)
- [MCP Tool Result Block](#)
- [Multiple MCP servers](#)
- [Authentication](#)
- [Obtaining an access token for testing](#)
- [Using the access token](#)

Remote MCP servers - Anthropic

Overview

Several companies have deployed remote MCP servers that developers can connect to via the Anthropic MCP connector API. These servers expand the capabilities available to developers and end users by providing remote access to various services and tools through the MCP protocol.

The remote MCP servers listed below are third-party services designed to work with the Anthropic API. These servers are not owned, operated, or endorsed by Anthropic. Users should only connect to remote MCP servers they trust and should review each server's security practices and terms before connecting.

Connecting to remote MCP servers

To connect to a remote MCP server:

1. Review the documentation for the specific server you want to use.
2. Ensure you have the necessary authentication credentials.
3. Follow the server-specific connection instructions provided by each company.

For more information about using remote MCP servers with the Anthropic API, see the [MCP connector docs](#).

Remote MCP server examples

Company	Description	Server URL
Asana	Interact with your Asana workspace through AI tools to keep projects on track.	<code>https://mcp.asana.com/sse</code>
Atlassian	Access Atlassian's collaboration and productivity tools.	<code>https://mcp.atlassian.com/v1/sse</code>
Cloudflare	Build applications, analyze traffic, monitor performance, and manage security settings through Cloudflare.	See mcp-server-cloudflare repo for server URLs
Intercom	Access real-time customer conversations, tickets, and user data—from Intercom.	<code>https://mcp.intercom.com/sse</code>
invideo	Build video creation capabilities into your applications.	<code>https://mcp.invideo.io/sse</code>
Linear	Integrate with Linear's issue tracking and project management system.	<code>https://mcp.linear.app/sse</code>
PayPal	Integrate PayPal commerce capabilities.	<code>https://mcp.paypal.com/sse</code>
Plaid	Analyze, troubleshoot, and optimize Plaid integrations.	<code>https://api.dashboard.plaid.com/mcp/sse</code>
Square	Use an agent to build on Square APIs. Payments, inventory, orders, and more.	<code>https://mcp.squareup.com/sse</code>
Workato	Access any application, workflows or data via Workato, made accessible for AI	MCP servers are programmatically generated.
Zapier	Connect to nearly 8,000 apps through Zapier's automation platform.	<code>https://mcp.zapier.com/</code>

Feedback

Was this page helpful?

Yes | No

[? Overview](#)

On this page

- [Connecting to remote MCP servers](#)
 - [Remote MCP server examples](#)
-

[Facebook](#) | [LinkedIn](#) | [Discord](#)

Guides to common use cases - Anthropic

Use cases

Guides to common use cases

Claude is designed to excel in a variety of tasks. Explore these in-depth production guides to learn how to build common use cases with Claude.

[Ticket routing](#)

Best practices for using Claude to classify and route customer support tickets at scale.

[Customer support agent](#)

Build intelligent, context-aware chatbots with Claude to enhance customer support interactions.

[Content moderation](#)

Techniques and best practices for using Claude to perform content filtering and general content moderation.

[Legal summarization](#)

Summarize legal documents using Claude to extract key information and expedite research.

Was this page helpful?

Yes No

[Remote MCP servers](#)

[Ticket routing](#)

Follow us:

[Twitter](#)

[LinkedIn](#)

[Discord](#)

Ticket routing - Anthropic

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```



Hello World

```
console.log("Hello")
```

Hello



- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Ticket routing - Anthropic

[Anthropic home page](#)

English

Search...

- [Research](#)
- [Login](#)
- [Sign up](#)
- [Sign up](#)

Search...

Navigation

Use cases

Ticket routing

Welcome

[Developer Guide](#)

[API Guide](#)

[Claude Code](#)

[Model Context Protocol \(MCP\)](#)

[Resources](#)

[Release Notes](#)

-

[Documentation](#)

-

[Developer Discord](#)

-

[Support](#)

First steps

- [Intro to Claude](#)
- [Get started](#)

Models & pricing

- [Models overview](#)
- [Choosing a model](#)
- [Migrating to Claude 4](#)
- [Model deprecations](#)
- [Pricing](#)

Learn about Claude

- [Building with Claude](#)
- [Features overview](#)
- [Context windows](#)
- [Glossary](#)

Capabilities

- [Prompt caching](#)
- [Extended thinking](#)
- [Streaming Messages](#)
- [Batch processing](#)
- [Citations](#)
- [Multilingual support](#)
- [Token counting](#)
- [Embeddings](#)
- [Vision](#)
- [PDF support](#)
- [Files API](#)
- [Search results](#)
- [Google Sheets add-on](#)

Tools

- [Overview](#)
- [How to implement tool use](#)
- [Token-efficient tool use](#)
- [Fine-grained tool streaming](#)
- [Bash tool](#)
- [Code execution tool](#)
- [Computer use tool](#)
- [Text editor tool](#)
- [Web search tool](#)

Model Context Protocol (MCP)

- [MCP connector](#)
- [Remote MCP servers](#)

Use cases

- [Overview](#)
- [Ticket routing](#)
- [Customer support agent](#)
- [Content moderation](#)
- [Legal summarization](#)

Prompt engineering

- [Overview](#)
- [Claude 4 best practices](#)
- [Prompt generator](#)
- [Use prompt templates](#)
- [Prompt improver](#)
- [Be clear and direct](#)
- [Use examples \(multishot prompting\)](#)
- [Let Claude think \(CoT\)](#)
- [Use XML tags](#)
- [Give Claude a role \(system prompts\)](#)
- [Prefill Claude's response](#)
- [Chain complex prompts](#)
- [Long context tips](#)
- [Extended thinking tips](#)

Test & evaluate

- [Define success criteria](#)
- [Develop test cases](#)
- [Using the Evaluation Tool](#)
- [Reducing latency](#)

Strengthen guardrails

- [Reduce hallucinations](#)
- [Increase output consistency](#)
- [Mitigate jailbreaks](#)
- [Streaming refusals](#)
- [Reduce prompt leak](#)
- [Keep Claude in character](#)

Legal center

- [Anthropic Privacy Policy](#)
- [Security and compliance](#)

Use cases

Ticket routing

Copy page

This guide walks through how to harness Claude's advanced natural language understanding capabilities to classify customer support tickets at scale based on customer intent, urgency, prioritization, customer profile, and more.

Define whether to use Claude for ticket routing

Understand your current support approach

Before diving into automation, it's crucial to understand your existing ticketing system. Start by investigating how your support team currently handles ticket routing.

Consider questions like:

- What criteria are used to determine what SLA/service offering is applied?
- Is ticket routing used to determine which tier of support or product specialist a ticket goes to?
- Are there any automated rules or workflows already in place? In what cases do they fail?
- How are edge cases or ambiguous tickets handled?
- How does the team prioritize tickets?

The more you know about how humans handle certain cases, the better you will be able to work with Claude to do the task.

Define user intent categories

A well-defined list of user intent categories is crucial for accurate support ticket classification with Claude. Claude's ability to route tickets effectively within your system is directly proportional to how well-defined your system's categories are.

Here are some example user intent categories and subcategories.

Technical issue

- Hardware problem
- Software bug
- Compatibility issue
- Performance problem

Account management

- Password reset
- Account access issues
- Billing inquiries
- Subscription changes

Product information

- Feature inquiries
- Product compatibility questions
- Pricing information
- Availability inquiries

User guidance

- How-to questions
- Feature usage assistance
- Best practices advice
- Troubleshooting guidance

Feedback

- Bug reports
- Feature requests
- General feedback or suggestions

- Complaints

Order-related

- Order status inquiries
- Shipping information
- Returns and exchanges
- Order modifications

Service request

- Installation assistance
- Upgrade requests
- Maintenance scheduling
- Service cancellation

Security concerns

- Data privacy inquiries
- Suspicious activity reports
- Security feature assistance

Compliance and legal

- Regulatory compliance questions
- Terms of service inquiries
- Legal documentation requests

Emergency support

- Critical system failures
- Urgent security issues
- Time-sensitive problems

Training and education

- Product training requests
- Documentation inquiries
- Webinar or workshop information

Integration and API

- Integration assistance
- API usage questions
- Third-party compatibility inquiries

In addition to intent, ticket routing and prioritization may also be influenced by other factors such as urgency, customer type, SLAs, or language. Be sure to consider other routing criteria when building your automated routing system.

Establish success criteria

Work with your support team to [define clear success criteria](#) with measurable benchmarks, thresholds, and goals.

Here are some standard criteria and benchmarks when using LLMs for support ticket routing:

Classification consistency

This metric assesses how consistently Claude classifies similar tickets over time. It's crucial for maintaining routing reliability. Measure this by periodically testing the model with a set of standardized inputs and aiming for a consistency rate of 95% or higher.

Adaptation speed

This measures how quickly Claude can adapt to new categories or changing ticket patterns. Test this by introducing new ticket types and measuring the time it takes for the model to achieve satisfactory accuracy (e.g., >90%) on these new categories. Aim for adaptation within 50-100 sample tickets.

Multilingual handling

This assesses Claude's ability to accurately route tickets in multiple languages. Measure the routing accuracy across different languages, aiming for no more than a 5-10% drop in accuracy for non-primary languages.

Edge case handling

This evaluates Claude's performance on unusual or complex tickets. Create a test set of edge cases and measure the routing accuracy, aiming for at least 80% accuracy on these challenging inputs.

Bias mitigation

This measures Claude's fairness in routing across different customer demographics. Regularly audit routing decisions for potential biases, aiming for consistent routing accuracy (within 2-3%) across all customer groups.

Prompt efficiency

In situations where minimizing token count is crucial, this criteria assesses how well Claude performs with minimal context. Measure routing accuracy with varying amounts of context provided, aiming for 90%+ accuracy with just the ticket title and a brief description.

Explainability score

This evaluates the quality and relevance of Claude's explanations for its routing decisions. Human raters can score explanations on a scale (e.g., 1-5), with the goal of achieving an average score of 4 or higher.

Here are some common success criteria that may be useful regardless of whether an LLM is used:

Routing accuracy

Routing accuracy measures how often tickets are correctly assigned to the appropriate team or individual on the first try. This is typically measured as a percentage of correctly routed tickets out of total tickets. Industry benchmarks often aim for 90-95% accuracy, though this can vary based on the complexity of the support structure.

Time-to-assignment

This metric tracks how quickly tickets are assigned after being submitted. Faster assignment times generally lead to quicker resolutions and improved customer satisfaction. Best-in-class systems often achieve average assignment times of under 5 minutes, with many aiming for near-instantaneous routing (which is possible with LLM implementations).

Rerouting rate

The rerouting rate indicates how often tickets need to be reassigned after initial routing. A lower rate suggests more accurate initial routing. Aim for a rerouting rate below 10%, with top-performing systems achieving rates as low as 5% or less.

First-contact resolution rate

This measures the percentage of tickets resolved during the first interaction with the customer. Higher rates indicate efficient routing and well-prepared support teams. Industry benchmarks typically range from 70-75%, with top performers achieving rates of 80% or higher.

Average handling time

Average handling time measures how long it takes to resolve a ticket from start to finish. Efficient routing can significantly reduce this time. Benchmarks vary widely by industry and complexity, but many organizations aim to keep average handling time under 24 hours for non-critical issues.

Customer satisfaction scores

Often measured through post-interaction surveys, these scores reflect overall customer happiness with the support process. Effective routing contributes to higher satisfaction. Aim for CSAT scores of 90% or higher, with top performers often achieving 95%+ satisfaction rates.

Escalation rate

This measures how often tickets need to be escalated to higher tiers of support. Lower escalation rates often indicate more accurate initial routing. Strive for an escalation rate below 20%, with best-in-class systems achieving rates of 10% or less.

Agent productivity

This metric looks at how many tickets agents can handle effectively after implementing the routing solution. Improved routing should increase productivity. Measure this by tracking tickets resolved per agent per day or hour, aiming for a 10-20% improvement after implementing a new routing system.

Self-service deflection rate

This measures the percentage of potential tickets resolved through self-service options before entering the routing system. Higher rates indicate effective pre-routing triage. Aim for a deflection rate of 20-30%, with top performers achieving rates of 40% or higher.

Cost per ticket

This metric calculates the average cost to resolve each support ticket. Efficient routing should help reduce this cost over time. While benchmarks vary widely, many organizations aim to reduce cost per ticket by 10-15% after implementing an improved routing system.

Choose the right Claude model

The choice of model depends on the trade-offs between cost, accuracy, and response time.

Many customers have found `claude-3-5-haiku-20241022` an ideal model for ticket routing, as it is the fastest and most cost-effective model in the Claude 3 family while still delivering excellent results. If your classification problem requires deep subject matter expertise or a large volume of intent categories complex reasoning, you may opt for the [larger Sonnet model](#).

Build a strong prompt

Ticket routing is a type of classification task. Claude analyzes the content of a support ticket and classifies it into predefined categories based on the issue type, urgency, required expertise, or other relevant factors.

Let's write a ticket classification prompt. Our initial prompt should contain the contents of the user request and return both the reasoning and the intent.

Try the [prompt generator](#) on the [Anthropic Console](#) to have Claude write a first draft for you.

Here's an example ticket routing classification prompt:

```
def classify_support_request(ticket_contents):
    # Define the prompt for the classification task
    classification_prompt = (
        f"You will be acting as a customer support ticket classification system."
        "Your task is to analyze customer support requests and output the appropriate "
        "for each request, along with your reasoning. "
        "Here is the customer support request you need to classify:"
        f"{ticket_contents}"
        "Please carefully analyze the above request to determine the customer's core i"
        "Consider what the customer is asking for and has concerns about. "
        "First, write out your reasoning and analysis of how to classify this request "
        "Then, output the appropriate classification label inside tags. "
        "The valid intents are: "
        " "
        "Support, Feedback, Complaint "
        "Order Tracking "
        "Refund/Exchange "
        " "
        "A request may have ONLY ONE applicable intent. "
        "Only include the intent that is most applicable to the request. "
        "As an example, consider the following request: "
        "Hello! I had high-speed fiber internet installed on Saturday, and my installe"
        "Where can I send my positive review? Thanks for your help! "
        "Here is an example of how your output should be formatted (for the above requ"
        "The user seeks information in order to leave positive feedback. "
        "Support, Feedback, Complaint"
    )
    # Send the prompt to the API to classify the support request.
    message = client.messages.create(
        model=DEFAULT_MODEL,
        max_tokens=500,
        temperature=0,
        messages=[{"role": "user", "content": classification_prompt}]
    )
    reasoning_and_intent = message.content[0].text
    # Extract reasoning using regex
    reasoning_match = re.search(r"(.*)", reasoning_and_intent, re.DOTALL)
    reasoning = reasoning_match.group(1).strip() if reasoning_match else ""
    # Extract intent using regex
    intent_match = re.search(r"(.*)", reasoning_and_intent, re.DOTALL)
    intent = intent_match.group(1).strip() if intent_match else ""
    return reasoning, intent
```

This code:

- Imports the Anthropic library and creates a client instance using your API key.
- Defines a `classify_support_request` function that takes a `ticket_contents` string.
- Sends the `ticket_contents` to Claude for classification using the `classification_prompt`
- Returns the model's `reasoning` and `intent` extracted from the response.

Since we need to wait for the entire reasoning and intent text to be generated before parsing, we set `stream=False` (the default).

Evaluate your prompt

Prompting often requires testing and optimization for it to be production ready. To determine the readiness of your solution, evaluate performance based on the success criteria and thresholds you established earlier.

To run your evaluation, you will need test cases to run it on. The rest of this guide assumes you have already [developed your test cases](#).

Build an evaluation function

Our example evaluation for this guide measures Claude's performance along three key metrics:

- Accuracy
- Cost per classification

You may need to assess Claude on other axes depending on what factors that are important to you.

To assess this, we first have to modify the script we wrote and add a function to compare the predicted intent with the actual intent and calculate the percentage of correct predictions. We also have to add in cost calculation and time measurement functionality.

```
import anthropic
import re
```

Create an instance of the Anthropic AI

```
client = anthropic.Anthropic()
```

Set the default model

```
DEFAULT_MODEL="claude-3-5-haiku-20241022"
```

```
def classify_support_request(request, actual_intent):
    # Define the prompt for the classification task
    classification_prompt = (
        f"You will be acting as a customer support ticket classification system."
        "Your task is to analyze customer support requests and output the appropriate"
        "for each request, along with your reasoning. "
        "Here is the customer support request you need to classify:"
        f"{request}"
        "Please carefully analyze the above request to determine the customer's core i"
        "Consider what the customer is asking for and has concerns about. "
```

```

        "First, write out your reasoning and analysis of how to classify this request
        "Then, output the appropriate classification label inside tags. "
        "The valid intents are: "
        " "
        "Support, Feedback, Complaint "
        "Order Tracking "
        "Refund/Exchange "
        " "
        "A request may have ONLY ONE applicable intent. "
        "Only include the intent that is most applicable to the request. "
        "As an example, consider the following request: "
        "Hello! I had high-speed fiber internet installed on Saturday, and my installe
        "Where can I send my positive review? Thanks for your help! "
        "Here is an example of how your output should be formatted (for the above requ
        "The user seeks information in order to leave positive feedback. "
        "Support, Feedback, Complaint"
    )
    # Send the prompt to the API to classify the support request.
    message = client.messages.create(
        model=DEFAULT_MODEL,
        max_tokens=500,
        temperature=0,
        messages=[{"role": "user", "content": classification_prompt}]
    )
    reasoning_and_intent = message.content[0].text
    # Extract reasoning using regex
    reasoning_match = re.search(r"(.*?)", reasoning_and_intent, re.DOTALL)
    reasoning = reasoning_match.group(1).strip() if reasoning_match else ""
    # Extract intent using regex
    intent_match = re.search(r"(.*?)", reasoning_and_intent, re.DOTALL)
    intent = intent_match.group(1).strip() if intent_match else ""
    # Check if the model's prediction is correct.
    correct = (actual_intent.strip() == intent.strip())
    # Return the reasoning, intent, correct, and usage.
    return reasoning, intent, correct, message.usage

```

Let's break down the edits we've made:

- We added the `actual_intent` from our test cases into the `classify_support_request` method and set up a comparison to assess whether Claude's intent classification matches our golden intent classification.
- We extracted usage statistics for the API call to calculate cost based on input and output tokens used

Run your evaluation

A proper evaluation requires clear thresholds and benchmarks to determine what is a good result. The script above will give us the runtime values for accuracy, response time, and cost per classification, but we still would need clearly established thresholds. For example:

- **Accuracy:** 95% (out of 100 tests)
- **Cost per classification:** 50% reduction on average (across 100 tests) from current routing method

Having these thresholds allows you to quickly and easily tell at scale, and with impartial empiricism, what method is best for you and what changes might need to be made to better fit your requirements.

Improve performance

In complex scenarios, it may be helpful to consider additional strategies to improve performance beyond standard [prompt engineering techniques](#) & [guardrail implementation strategies](#). Here are some common scenarios:

Use a taxonomic hierarchy for cases with 20+ intent categories

As the number of classes grows, the number of examples required also expands, potentially making the prompt unwieldy. As an alternative, you can consider implementing a hierarchical classification system using a mixture of classifiers.

1. Organize your intents in a taxonomic tree structure.
2. Create a series of classifiers at every level of the tree, enabling a cascading routing approach.

For example, you might have a top-level classifier that broadly categorizes tickets into “Technical Issues,” “Billing Questions,” and “General Inquiries.” Each of these categories can then have its own sub-classifier to further refine the classification.

- **Pros - greater nuance and accuracy:** You can create different prompts for each parent path, allowing for more targeted and context-specific classification. This can lead to improved accuracy and more nuanced handling of customer requests.
- **Cons - increased latency:** Be advised that multiple classifiers can lead to increased latency, and we recommend implementing this approach with our fastest model, Haiku.

Use vector databases and similarity search retrieval to handle highly variable tickets

Despite providing examples being the most effective way to improve performance, if support requests are highly variable, it can be hard to include enough examples in a single prompt.

In this scenario, you could employ a vector database to do similarity searches from a dataset of examples and retrieve the most relevant examples for a given query.

This approach, outlined in detail in our [classification recipe](#), has been shown to improve performance from 71% accuracy to 93% accuracy.

Account specifically for expected edge cases

Here are some scenarios where Claude may misclassify tickets (there may be others that are unique to your situation). In these scenarios, consider providing explicit instructions or examples in the prompt of how Claude should handle the edge case:

Customers make implicit requests

Customers often express needs indirectly. For example, “I’ve been waiting for my package for over two weeks now” may be an indirect request for order status.

- **Solution:** Provide Claude with some real customer examples of these kinds of requests, along with what the underlying intent is. You can get even better results if you include a classification rationale for

particularly nuanced ticket intents, so that Claude can better generalize the logic to other tickets.

Claude prioritizes emotion over intent

When customers express dissatisfaction, Claude may prioritize addressing the emotion over solving the underlying problem.

- **Solution:** Provide Claude with directions on when to prioritize customer sentiment or not. It can be something as simple as “Ignore all customer emotions. Focus only on analyzing the intent of the customer’s request and what information the customer might be asking for.”

Multiple issues cause issue prioritization confusion

When customers present multiple issues in a single interaction, Claude may have difficulty identifying the primary concern.

- **Solution:** Clarify the prioritization of intents so that Claude can better rank the extracted intents and identify the primary concern.
-

Integrate Claude into your greater support workflow

Proper integration requires that you make some decisions regarding how your Claude-based ticket routing script fits into the architecture of your greater ticket routing system. There are two ways you could do this:

- **Push-based:** The support ticket system you’re using (e.g., Zendesk) triggers your code by sending a webhook event to your routing service, which then classifies the intent and routes it.
 - This approach is more web-scalable, but needs you to expose a public endpoint.
- **Pull-Based:** Your code pulls for the latest tickets based on a given schedule and routes them at pull time.
 - This approach is easier to implement but might make unnecessary calls to the support ticket system when the pull frequency is too high or might be overly slow when the pull frequency is too low.

For either approach, you will need to wrap your script in a service. The choice depends on what APIs your support system provides.

Classification cookbook

[Visit our classification cookbook for more example code and detailed eval guidance.](#)

Anthropic Console

[Begin building and evaluating your workflow on the Anthropic Console.](#)

Was this page helpful?

Yes No

[Overview Customer support agent](#)

[x linkedin discord](#)

On this page

- [Define whether to use Claude for ticket routing](#)
- [Build and deploy your LLM support workflow](#)
- [Understand your current support approach](#)
- [Define user intent categories](#)
- [Establish success criteria](#)

- Choose the right Claude model
- Build a strong prompt
- Deploy your prompt
- Evaluate your prompt
- Build an evaluation function
- Run your evaluation
- Improve performance
- Use a taxonomic hierarchy for cases with 20+ intent categories
- Use vector databases and similarity search retrieval to handle highly variable tickets
- Account specifically for expected edge cases
- Integrate Claude into your greater support workflow

Customer support agent - Anthropic

Introduction

This guide walks through how to leverage Claude's advanced conversational capabilities to handle customer inquiries in real time, providing 24/7 support, reducing wait times, and managing high support volumes with accurate responses and positive interactions.

Before building with Claude

Decide whether to use Claude for support chat

Here are some key indicators that you should employ an LLM like Claude to automate portions of your customer support process:

- **High volume of repetitive queries**
Claude excels at handling a large number of similar questions efficiently, freeing up human agents for more complex issues.
- **Need for quick information synthesis**
Claude can quickly retrieve, process, and combine information from vast knowledge bases, while human agents may need time to research or consult multiple sources.
- **24/7 availability requirement**
Claude can provide round-the-clock support without fatigue, whereas staffing human agents for continuous coverage can be costly and challenging.
- **Rapid scaling during peak periods**
Claude can handle sudden increases in query volume without the need for hiring and training additional staff.
- **Consistent brand voice**
You can instruct Claude to consistently represent your brand's tone and values, whereas human agents may vary in their communication styles.

Some considerations for choosing Claude over other LLMs:

- **You prioritize natural, nuanced conversation:** Claude's sophisticated language understanding allows for more natural, context-aware conversations that feel more human-like than chats with other LLMs.
- **You often receive complex and open-ended queries:** Claude can handle a wide range of topics and inquiries without generating canned responses or requiring extensive programming of permutations of user utterances.
- **You need scalable multilingual support:** Claude's multilingual capabilities allow it to engage in conversations in over 200 languages without the need for separate chatbots or extensive translation processes for each supported language.

Define your ideal chat interaction

Outline an ideal customer interaction to define how and when you expect the customer to interact with Claude. This outline will help to determine the technical requirements of your solution.

Example chat interaction for car insurance customer support:

- **Customer:** Initiates support chat experience
Claude: Warmly greets customer and initiates conversation
- **Customer:** Asks about insurance for their new electric car
Claude: Provides relevant information about electric vehicle coverage
- **Customer:** Asks questions related to unique needs for electric vehicle insurances
Claude: Responds with accurate and informative answers and provides links to the sources
- **Customer:** Asks off-topic questions unrelated to insurance or cars
Claude: Clarifies it does not discuss unrelated topics and steers the user back to car insurance
- **Customer:** Expresses interest in an insurance quote
Claude: Asks questions to determine the quote, then calls API tool, synthesizes response, and presents quote
- **Customer:** Asks follow-up questions
Claude: Answers and guides next steps, closing conversation

In your own use case, writing out actual words can help you get the tone, response length, and detail level right.

Break the interaction into tasks

Customer support involves multiple tasks:

1. **Greeting and Guidance**
2. **Product Information**
3. **Conversation Management**
4. **Quote Generation**

Ensure you prompt and evaluate each task separately for better coverage and testing.

Establish success criteria

Work with your team to define measurable benchmarks:

- **Query comprehension accuracy** ($\geq 95\%$)
- **Response relevance** ($\geq 90\%$)
- **Response correctness** (100%) on info given to Claude
- **Source citation relevance** ($\geq 80\%$)
- **Topic adherence** ($\geq 95\%$)
- **Content generation success** (100% when needed)
- **Escalation accuracy** ($\geq 95\%$)

Business impact benchmarks:

- **Customer sentiment maintained/improved in 90%**
- **70-80% inquiry deflection**
- **CSAT of 4/5 or higher**
- **Lower average handle time**

How to implement Claude as a customer support agent

Choose the right Claude model

Balancing cost, accuracy, response time:

- `claude-opus-4-20250514` is good for general support
- `claude-3-haiku-20240307` is better for latency-critical workflows with multiple prompts

Build a strong prompt

Define Claude's role and context clearly:

```
IDENTITY = "You are Eva, a friendly and knowledgeable AI assistant for Acme Insurance.
```

Break down prompts into subsections for easier debugging and adaptation.

Sample static content

Use static contexts for consistent information. Example:

```
STATIC_GREETINGS_AND_GENERAL = """
<static_context>
Acme Auto Insurance: Your Trusted Companion on the Road...
[full content as above]
</static_context>
"""
```

Similarly, prepare static info for car insurance and electric vehicle coverage.

Sample interactions

Provide 4-5 representative dialogue examples illustrating ideal customer interactions, guardrails, and tool calls.

Additional guardrails

```
ADDITIONAL_GUARDRAILS = """
1. Only provide info about listed insurance types.
2. Politely state if ask about unsupported types.
3. Do not speculate on future products.
4. Do not make unapproved promises.
5. Do not mention competitors.
"""
```

Combining instructions

Create a comprehensive prompt string:

```
TASK_SPECIFIC_INSTRUCTIONS = " ".join([
    STATIC_GREETINGS_AND_GENERAL,
    STATIC_CAR_INSURANCE,
    STATIC_ELECTRIC_CAR_INSURANCE,
    EXAMPLES,
    ADDITIONAL_GUARDRAILS
])
```

Add dynamic and agentic capabilities with tool use

Start with listing tools, e.g., a quote calculator:

```
TOOLS = [  
    {  
        "name": "get_quote",  
        "description": "Calculate the insurance quote based on user input. Returned value  
        "input_schema": {  
            "type": "object",  
            "properties": {  
                "make": {"type": "string", "description": "The make of the vehicle."},  
                "model": {"type": "string", "description": "The model of the vehicle."},  
                "year": {"type": "integer", "description": "Model year."},  
                "mileage": {"type": "integer", "description": "Vehicle mileage."},  
                "driver_age": {"type": "integer", "description": "Primary driver age."}  
            },  
            "required": ["make", "model", "year", "mileage", "driver_age"]  
        }  
    }  
]
```

Define function to call:

```
def get_quote(make, model, year, mileage, driver_age):  
    # Simulate API call or retrieval process  
    time.sleep(1)  
    return 100 # Fixed sample for illustration
```

Deploy your prompts

Create a `chatbot.py` with a class encapsulating interactions, e.g.:

```
from anthropic import Anthropic  
from config import IDENTITY, TOOLS, MODEL, get_quote  
from dotenv import load_dotenv  
  
load_dotenv()  
  
class ChatBot:  
    def __init__(self, session_state):  
        self.anthropic = Anthropic()  
        self.session_state = session_state  
  
    def generate_message(self, messages, max_tokens):  
        # call Anthropic API  
        ...  
  
    def process_user_input(self, user_input):  
        # Append user message, generate response, handle tools  
        ...
```

Use Streamlit for interface in `app.py`, managing conversation state, displaying messages, and processing user input.

Evaluate your prompts

Use systematic testing, including the [Evaluation tool](#), to refine performance against success criteria.

Improve performance

Implement advanced techniques:

- **Reduce long context latency with RAG**, using embeddings to retrieve relevant info dynamically.
- **Integrate real-time data with tool use** for live info like account status.
- **Strengthen input/output guardrails**: Fact-checking, citations, avoiding commitments.
- **Reduce perceived latency with streaming responses**.
- **Scale architecture** with prompt engineering, tools, or intent classifiers.

Integrate Claude into your workflow

Wrap via an API (Flask, FastAPI), support streaming via SSE, ensure context retention, and build web UI.

[Retrieval Augmented Generation \(RAG\) cookbook](#)

[Citations cookbook](#)

Feedback

Was this page helpful?

[Yes](#), [No](#)

On this page

- [Before building with Claude](#)
- [Decide whether to use Claude for support chat](#)
- [Define your ideal chat interaction](#)
- [Break the interaction into tasks](#)
- [Establish success criteria](#)
- [Implement as a customer service agent](#)
- [Choose the appropriate model](#)
- [Build a strong prompt](#)
- [Add capabilities with tool use](#)
- [Deploy prompts](#)
- [Create user interface](#)
- [Evaluate prompts](#)
- [Improve performance](#)
- [Reduce latency with RAG](#)
- [Integrate real-time data](#)
- [Strengthen guardrails](#)
- [Reduce perceived response time](#)
- [Scale your chatbot](#)
- [Integrate Claude into workflow](#)

Content moderation - Anthropic

Use cases

Content moderation

Content moderation is a critical aspect of maintaining a safe, respectful, and productive environment in digital applications. In this guide, we'll discuss how Claude can be used to moderate content within your digital application.

Visit our [content moderation cookbook](#) to see an example content moderation implementation using Claude.

This guide is focused on moderating user-generated content within your application. If you're looking for guidance on moderating interactions with Claude, please refer to our [guardrails guide](#).

Before building with Claude

Decide whether to use Claude for content moderation

Here are some key indicators that you should use an LLM like Claude instead of a traditional ML or rules-based approach for content moderation:

- **You want a cost-effective and rapid implementation**
Traditional ML methods require significant engineering resources, ML expertise, and infrastructure costs. Human moderation systems incur even higher costs. With Claude, you can have a sophisticated moderation system up and running in a fraction of the time for a fraction of the price.
- **You desire both semantic understanding and quick decisions**
Traditional ML approaches, such as bag-of-words models or simple pattern matching, often struggle to understand the tone, intent, and context of the content. While human moderation systems excel at understanding semantic meaning, they require time for content to be reviewed. Claude bridges the gap by combining semantic understanding with the ability to deliver moderation decisions quickly.
- **You need consistent policy decisions**
By leveraging its advanced reasoning capabilities, Claude can interpret and apply complex moderation guidelines uniformly. This consistency helps ensure fair treatment of all content, reducing the risk of inconsistent or biased moderation decisions that can undermine user trust.
- **Your moderation policies are likely to change or evolve over time**
Once a traditional ML approach has been established, changing it is a laborious and data-intensive undertaking. On the other hand, as your product or customer needs evolve, Claude can easily adapt to changes or additions to moderation policies without extensive relabeling of training data.
- **You require interpretable reasoning for your moderation decisions**
If you wish to provide users or regulators with clear explanations behind moderation decisions, Claude can generate detailed and coherent justifications. This transparency is important for building trust and ensuring accountability in content moderation practices.
- **You need multilingual support without maintaining separate models**
Traditional ML approaches typically require separate models or extensive translation processes for each supported language. Human moderation requires hiring a workforce fluent in each supported language. Claude's multilingual capabilities allow it to classify tickets in various languages without the need for separate models or extensive translation processes, streamlining moderation for global customer bases.

- **You require multimodal support**

Claude’s multimodal capabilities allow it to analyze and interpret content across both text and images. This makes it a versatile tool for comprehensive content moderation in environments where different media types need to be evaluated together.

Anthropic has trained all Claude models to be honest, helpful and harmless. This may result in Claude moderating content deemed particularly dangerous (in line with our [Acceptable Use Policy](#)), regardless of the prompt used. For example, an adult website that wants to allow users to post explicit sexual content may find that Claude still flags explicit content as requiring moderation, even if they specify in their prompt not to moderate explicit sexual content. We recommend reviewing our AUP in advance of building a moderation solution.

Generate examples of content to moderate

Before developing a content moderation solution, first create examples of content that should be flagged and content that should not be flagged. Ensure that you include edge cases and challenging scenarios that may be difficult for a content moderation system to handle effectively. Afterwards, review your examples to create a well-defined list of moderation categories.

For instance, the examples generated by a social media platform might include the following:

```
allowed_user_comments = [
    'This movie was great, I really enjoyed it. The main actor really killed it!',
    'I hate Mondays.',
    'It is a great time to invest in gold!'
]

disallowed_user_comments = [
    'Delete this post now or you better hide. I am coming after you and your family.',
    'Stay away from the 5G cellphones!! They are using 5G to control you.',
    'Congratulations! You have won a $1,000 gift card. Click here to claim your prize!'
]

user_comments = allowed_user_comments + disallowed_user_comments

# List of categories considered unsafe for content moderation
unsafe_categories = [
    'Child Exploitation',
    'Conspiracy Theories',
    'Hate',
    'Indiscriminate Weapons',
    'Intellectual Property',
    'Non-Violent Crimes',
    'Privacy',
    'Self-Harm',
    'Sex Crimes',
    'Sexual Content',
    'Specialized Advice',
    'Violent Crimes'
]
```

Effectively moderating these examples requires a nuanced understanding of language. In the comment `This movie was great, I really enjoyed it. The main actor really killed it!`, the content moderation system needs to recognize that “killed it” is a metaphor, not an indication of actual violence. Conversely, despite the lack of explicit mentions of violence, the comment `Delete this post now or you better hide. I am coming after you and your family.` should be flagged by the content moderation system.

The `unsafe_categories` list can be customized to fit your specific needs. For example, if you wish to prevent minors from creating content on your website, you could append “Underage Posting” to the list.

How to moderate content using Claude

Select the right Claude model

When selecting a model, it’s important to consider the size of your data. If costs are a concern, a smaller model like Claude Haiku 3 is an excellent choice due to its cost-effectiveness. Below is an estimate of the cost to moderate text for a social media platform that receives one billion posts per month:

- **Content size**
 - Posts per month: 1bn
 - Characters per post: 100
 - Total characters: 100bn
- **Estimated tokens**
 - Input tokens: 28.6bn (assuming 1 token per 3.5 characters)
 - Percentage of messages flagged: 3%
 - Output tokens per flagged message: 50
 - Total output tokens: 1.5bn
- **Claude Haiku 3 estimated cost**
 - Input token cost: $28.6bn * \$0.25/MTok \approx \715 million
 - Output token cost: $1.5bn * \$1.25/MTok \approx \1.875 billion
 - Monthly cost: approximately **\$2.59 million**
- **Claude Sonnet 4 estimated cost**
 - Input token cost: $28.6bn * \$3.00/MTok \approx \8.58 billion
 - Output token cost: $1.5bn * \$15.00/MTok \approx \22.5 billion
 - Monthly cost: approximately **\$31.08 million**

Actual costs may differ from these estimates. These estimates are based on the prompt highlighted in the section on [batch processing](#). Output tokens can be reduced even further by removing the `explanation` field from the response.

Build a strong prompt

In order to use Claude for content moderation, Claude must understand the moderation requirements of your application. Let’s start by writing a prompt that allows you to define your moderation needs:

```
import anthropic
import json

# Initialize the Anthropic client
client = anthropic.Anthropic()

def moderate_message(message, unsafe_categories):
    # Convert the list of unsafe categories into a string, with each category on a new
    unsafe_category_str = '\n'.join(unsafe_categories)

    # Construct the prompt for Claude, including the message and unsafe categories
    assessment_prompt = f"""
```

Determine whether the following message warrants moderation, based on the unsafe c

Message:

```
<message>{message}</message>
```

Unsafe Categories:

```
<categories>
{unsafe_category_str}
</categories>
```

Respond ONLY with a JSON object, using the format below:

```
{
  "violation": <Boolean denoting whether the message should be moderated>,
  "categories": [<list of violated categories>],
  "explanation": [<Optional. Explanation of why there's a violation>]
}
```

```
"""
# Send the request to Claude for content moderation
response = client.messages.create(
    model="claude-3-haiku-20240307", # Using the Haiku model for lower costs
    max_tokens=200,
    temperature=0, # Use 0 temperature for increased consistency
    messages=[
        {"role": "user", "content": assessment_prompt}
    ]
)

# Parse the JSON response from Claude
assessment = json.loads(response.content[0].text)

# Extract the violation status from the assessment
contains_violation = assessment['violation']
violated_categories = assessment.get('categories', []) if contains_violation else
explanation = assessment.get('explanation') if contains_violation else None

return contains_violation, violated_categories, explanation

# Process each comment and print the results
for comment in user_comments:
    print(f"\nComment: {comment}")
    violation, violated_categories, explanation = moderate_message(comment, unsafe_cat

    if violation:
        print(f"Violated Categories: {' , '.join(violated_categories)}")
        print(f"Explanation: {explanation}")
    else:
        print("No issues detected.")
```

This code implements a `moderate_message` function that uses Claude to evaluate the content of a message based on predefined unsafe categories. The prompt instructs Claude to respond with a JSON object detailing whether the message violates policies, which categories are involved, and an optional explanation.

Evaluate your prompt

Content moderation is a classification problem. Thus, you can use the same techniques outlined in our [classification cookbook](#) to determine the accuracy of your content moderation system.

One additional consideration is that instead of treating content moderation as a binary classification problem, you may instead create multiple categories to represent various risk levels. Creating multiple risk levels allows you to adjust the aggressiveness of your moderation. For example, you might want to automatically block user queries that are deemed high risk, while users with many medium risk queries are flagged for human review.

```
import anthropic
import json

# Initialize the Anthropic client
client = anthropic.Anthropic()

def assess_risk_level(message, unsafe_categories):
    # Convert the list of unsafe categories into a string, with each category on a new
    unsafe_category_str = '\n'.join(unsafe_categories)

    # Construct the prompt for Claude, including the message, unsafe categories, and r
    assessment_prompt = f"""
    Assess the risk level of the following message warrants moderation, based on the u

    Message:
    <message>{message}</message>

    Unsafe Categories:
    <categories>
    {unsafe_category_str}
    </categories>

    Assign a risk level based on your confidence that the user's message should be mod

    0 - No risk
    1 - Low risk
    2 - Medium risk
    3 - High risk

    Respond ONLY with a JSON object, using the format below:
    {{
      "risk_level": <Number denoting risk level>,
      "categories": [<list of violated categories>],
      "explanation": <Optional, explanation if risk_level > 0>
    }}
    """

    # Send the request to Claude for risk assessment
    response = client.messages.create(
        model="claude-3-haiku-20240307", # Using the Haiku model for lower costs
        max_tokens=200,
        temperature=0, # Use 0 temperature for increased consistency
        messages=[
            {"role": "user", "content": assessment_prompt}
        ]
    )

    # Parse the JSON response from Claude
```

```

assessment = json.loads(response.content[0].text)
risk_level = assessment['risk_level']
violated_categories = assessment['categories']
explanation = assessment.get('explanation')

return risk_level, violated_categories, explanation

# Process each comment and print the results
for comment in user_comments:
    print(f"\nComment: {comment}")
    risk_level, violated_categories, explanation = assess_risk_level(comment, unsafe_c
    print(f"Risk Level: {risk_level}")
    if violated_categories:
        print(f"Violated Categories: {' , '.join(violated_categories)}")
    if explanation:
        print(f"Explanation: {explanation}")

```

This approach allows nuanced moderation by assessing risk levels rather than binary violations, enabling more flexible moderation strategies.

Deploy your prompt

Once you are confident in the quality of your solution, it's time to deploy it to production. Here are some best practices to follow when using content moderation in production:

1. Provide clear feedback to users

When user input is blocked or a response is flagged due to content moderation, provide informative and constructive feedback to help users understand why their message was flagged and how they can rephrase it appropriately. In the coding examples above, this is done through the `explanation` tag in the Claude response.

2. Analyze moderated content

Keep track of the types of content being flagged by your moderation system to identify trends and potential areas for improvement.

3. Continuously evaluate and improve

Regularly assess the performance of your content moderation system using metrics such as precision and recall tracking. Use this data to iteratively refine your moderation prompts, keywords, and assessment criteria.

Improve performance

In complex scenarios, it may be helpful to consider additional strategies to improve performance beyond standard [prompt engineering techniques](#). Here are some advanced strategies:

Define topics and provide examples

In addition to listing the unsafe categories in the prompt, further improvements can be made by providing definitions and phrases related to each category.

```

import anthropic
import json

# Initialize the Anthropic client

```

```

client = anthropic.Anthropic()

# Dictionary of categories considered unsafe for content moderation, with their defini
unsafe_category_definitions = {
    'Child Exploitation': 'Content that depicts child nudity or that enables, encourag
    'Conspiracy Theories': 'Content that promotes or endorses unfounded, false, or mis
    'Hate': 'Content that is hateful toward people on the basis of their protected cha
    'Indiscriminate Weapons': 'Content that enables the creation of weapons of mass de
    'Intellectual Property': 'Content that violates the intellectual property rights o
    'Non-Violent Crimes': 'Content that enables, encourages, or excuses the commission
    'Privacy': 'Content that contains sensitive, personal information about private in
    'Self-Harm': 'Content that enables, encourages, or excuses acts of intentional sel
    'Sex Crimes': 'Content that enables, encourages, or excuses the commission of sex
    'Sexual Content': 'Content that contains erotic descriptions or explicit depiction
    'Specialized Advice': 'Content that contains financial, medical, or legal advice.
    'Violent Crimes': 'Content that enables, encourages, or excuses the commission of
}

def moderate_message_with_definitions(message, unsafe_category_definitions):
    # Format unsafe categories string, with each category and its definition on a new
    unsafe_category_str = '\n'.join(f'{category}: {definition}' for category, definiti

    # Construct the prompt for Claude, including the message and unsafe categories
    assessment_prompt = f"""
    Determine whether the following message warrants moderation, based on the unsafe c

    Message:
    <message>{message}</message>

    Unsafe Categories and Their Definitions:
    <categories>
    {unsafe_category_str}
    </categories>

    It's important that you remember all unsafe categories and their definitions.

    Respond ONLY with a JSON object, using the format below:
    {{
        "violation": <Boolean>,
        "categories": [<list of violated categories>],
        "explanation": <Optional, explanation if violation>
    }}
    """

    # Send request to Claude
    response = client.messages.create(
        model="claude-3-haiku-20240307",
        max_tokens=200,
        temperature=0,
        messages=[
            {"role": "user", "content": assessment_prompt}
        ]
    )

```

```
assessment = json.loads(response.content[0].text)
return assessment
```

This expanded approach ensures each category is accompanied by its clear definition, leading to more precise moderation.

Consider batch processing

To reduce costs in situations where real-time moderation isn't necessary, consider moderating messages in batches. Include multiple messages within the prompt's context, and ask Claude to assess which messages should be moderated.

```
import anthropic
import json

# Initialize the Anthropic client
client = anthropic.Anthropic()

def batch_moderate_messages(messages, unsafe_categories):
    # Convert the list of unsafe categories into a string, with each category on a new
    unsafe_category_str = '\n'.join(unsafe_categories)

    # Format messages string, with each message wrapped in XML-like tags and given an
    messages_str = '\n'.join(f'<message id={idx}>{msg}</message>' for idx, msg in enum

    # Construct the prompt
    assessment_prompt = f"""
    Determine the messages to moderate, based on the unsafe categories outlined below.

    Messages:
    {messages_str}
    </messages>

    Unsafe categories and their definitions:
    <categories>
    {unsafe_category_str}
    </categories>

    Respond ONLY with a JSON object, listing all message IDs that violate policies:

    {{
      "violations": [
        {{
          "id": <message id>,
          "categories": [<list of violated categories>],
          "explanation": <Explanation of why there's a violation>
        }},
        ...
      ]
    }}

    # Remember to analyze every message for violations.
    # Select any violations that reasonably apply.
    """
```



```

# Send request to Claude
response = client.messages.create(
    model="claude-3-haiku-20240307",
    max_tokens=2048,
    temperature=0,
    messages=[
        {"role": "user", "content": assessment_prompt}
    ]
)

assessment = json.loads(response.content[0].text)
return assessment

# Example usage:
# response_obj = batch_moderate_messages(user_comments, unsafe_categories)

# Print detected violations
# for violation in response_obj['violations']:
#     print(f"Comment ID: {violation['id']}")
#     print(f"Violated Categories: {' , '.join(violation['categories'])}")
#     print(f"Explanation: {violation['explanation']}")

```

Batch moderation helps optimize cost-efficiency in large-scale applications, with the tradeoff of some potential accuracy reduction depending on batch size. Adjust `max_tokens` accordingly.

Additional Resources

- [Content moderation cookbook](#): See a fully implemented code example for content moderation.
- [Guardrails guide](#): Techniques to improve interaction safety with Claude.

Legal summarization - Anthropic

Use cases

Legal summarization

Copy page

This guide walks through how to leverage Claude's advanced natural language processing capabilities to efficiently summarize legal documents, extracting key information and expediting legal research. With Claude, you can streamline the review of contracts, litigation prep, and regulatory work, saving time and ensuring accuracy in your legal processes.

Visit our [summarization cookbook](#) to see an example legal summarization implementation using Claude.

Before building with Claude

Decide whether to use Claude for legal summarization

Here are some key indicators that you should employ an LLM like Claude to summarize legal documents:

- You want to review a high volume of documents efficiently and affordably
Large-scale document review can be time-consuming and expensive when done manually. Claude can process and summarize vast amounts of legal documents rapidly, significantly reducing the time and cost associated with document review. This capability is particularly valuable for tasks like due diligence, contract analysis, or litigation discovery, where efficiency is crucial.
- You require automated extraction of key metadata
Claude can efficiently extract and categorize important metadata from legal documents, such as parties involved, dates, contract terms, or specific clauses. This automated extraction can help organize information, making it easier to search, analyze, and manage large document sets. It's especially useful for contract management, compliance checks, or creating searchable databases of legal information.
- You want to generate clear, concise, and standardized summaries
Claude can generate structured summaries that follow predetermined formats, making it easier for legal professionals to quickly grasp the key points of various documents. These standardized summaries can improve readability, facilitate comparison between documents, and enhance overall comprehension, especially when dealing with complex legal language or technical jargon.
- You need precise citations for your summaries
When creating legal summaries, proper attribution and citation are crucial to ensure credibility and compliance with legal standards. Claude can be prompted to include accurate citations for all referenced legal points, making it easier for legal professionals to review and verify the summarized information.
- You want to streamline and expedite your legal research process
Claude can assist in legal research by quickly analyzing large volumes of case law, statutes, and legal commentary. It can identify relevant precedents, extract key legal principles, and summarize complex legal arguments. This capability can significantly speed up the research process, allowing legal professionals to focus on higher-level analysis and strategy development.

Determine the details you want the summarization to extract

There is no single correct summary for any given document. Without clear direction, it can be difficult for Claude to determine which details to include. To achieve optimal results, identify the specific information you want to include in the summary.

For instance, when summarizing a sublease agreement, you might wish to extract the following key points:

```
details_to_extract = [  
    'Parties involved (sublessor, sublessee, and original lessor)',  
    'Property details (address, description, and permitted use)',  
    'Term and rent (start date, end date, monthly rent, and security deposit)',  
    'Responsibilities (utilities, maintenance, and repairs)',  
    'Consent and notices (landlord\'s consent, and notice requirements)',  
    'Special provisions (furniture, parking, and subletting restrictions)'  
]
```

Establish success criteria

Evaluating the quality of summaries is a notoriously challenging task. Unlike many other natural language processing tasks, evaluation of summaries often lacks clear-cut, objective metrics. The process can be highly subjective, with different readers valuing different aspects of a summary. Here are criteria you may wish to consider when assessing how well Claude performs legal summarization:

- **Factual correctness:** The summary should accurately represent the facts, legal concepts, and key points in the document.
- **Legal precision:** Terminology and references to statutes, case law, or regulations must be correct and aligned with legal standards.
- **Conciseness:** The summary should condense the legal document to its essential points without losing important details.
- **Consistency:** If summarizing multiple documents, the LLM should maintain a consistent structure and approach to each summary.
- **Readability:** The text should be clear and easy to understand. If the audience is not legal experts, the summarization should not include legal jargon that could confuse the audience.
- **Bias and fairness:** The summary should present an unbiased and fair depiction of the legal arguments and positions.

See our guide on [establishing success criteria](#) for more information.

How to summarize legal documents using Claude

Select the right Claude model

Model accuracy is extremely important when summarizing legal documents. Claude Sonnet 3.5 is an excellent choice for use cases such as this where high accuracy is required. If the size and quantity of your documents is large such that costs start to become a concern, you can also try using a smaller model like Claude Haiku 3.

To help estimate these costs, below is a comparison of the cost to summarize 1,000 sublease agreements using both Sonnet and Haiku:

- **Content size**
 - Number of agreements: 1,000
 - Characters per agreement: 300,000
 - Total characters: 300M
- **Estimated tokens**

- Input tokens: 86M (assuming 1 token per 3.5 characters)
- Output tokens per summary: 350
- Total output tokens: 350,000

- **Claude Sonnet 4 estimated cost**

- Input token cost: 86 MTok * \$3.00/MTok = \$258
- Output token cost: 0.35 MTok * \$15.00/MTok = \$5.25
- Total cost: \$263.25

- **Claude Haiku 3 estimated cost**

- Input token cost: 86 MTok * \$0.25/MTok = \$21.50
- Output token cost: 0.35 MTok * \$1.25/MTok = \$0.44
- Total cost: \$21.96

Actual costs may differ. These estimates are based on the example highlighted in the [prompting](#).

Transform documents into a format that Claude can process

Before you begin summarizing documents, you need to prepare your data. This involves extracting text from PDFs, cleaning the text, and ensuring it's ready to be processed by Claude.

Here is a demonstration on a sample PDF:

```
from io import BytesIO
import re
import pypdf
import requests

def get_llm_text(pdf_file):
    reader = pypdf.PdfReader(pdf_file)
    text = "\n".join([page.extract_text() for page in reader.pages])

    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text)

    # Remove page numbers
    text = re.sub(r'\n\s*\d+\s*\n', '\n', text)

    return text

# Create the full URL from the GitHub repository
url = "https://raw.githubusercontent.com/anthropics/anthropic-cookbook/main/skills/sum
url = url.replace(" ", "%20")

# Download the PDF file into memory
response = requests.get(url)

# Load the PDF from memory
pdf_file = BytesIO(response.content)

document_text = get_llm_text(pdf_file)
print(document_text[:50000])
```

In this example, we first download a pdf of a sample sublease agreement used in the [summarization cookbook](#). This agreement was sourced from a publicly available sublease agreement from the [sec.gov website](#).

We use the pypdf library to extract the contents of the pdf and convert it to text. The text data is then cleaned by removing extra whitespace and page numbers.

Build a strong prompt

Claude can adapt to various summarization styles. You can change the details of the prompt to guide Claude to be more or less verbose, include more or less technical terminology, or provide a higher or lower level summary of the context at hand.

Here's an example of how to create a prompt that ensures the generated summaries follow a consistent structure when analyzing sublease agreements:

```
import anthropic

# Initialize the Anthropic client
client = anthropic.Anthropic()

def summarize_document(text, details_to_extract, model="claude-opus-4-20250514", max_t

    # Format the details to extract to be placed within the prompt's context
    details_to_extract_str = '\n'.join(details_to_extract)

    # Prompt the model
    prompt = f"""Summarize the following sublease agreement. Focus on these key aspect

{details_to_extract_str}

Provide the summary in bullet points nested within the XML header for each section. Fo

<parties involved>
- Sublessor: [Name]
<!-- Add more details as needed -->
</parties involved>

If any information is not explicitly stated in the document, note it as "Not specified

Sublease agreement text:
{ text }
"""

    response = client.messages.create(
        model=model,
        max_tokens=max_tokens,
        system="You are a legal analyst specializing in real estate law, known for hig
        messages=[
            {"role": "user", "content": prompt},
            {"role": "assistant", "content": "Here is the summary of the sublease agre
        ],
        stop_sequences=["</summary>"]
    )

    return response.content[0].text

sublease_summary = summarize_document(document_text, details_to_extract)
print(sublease_summary)
```

This code implements a `summarize_document` function that uses Claude to summarize the contents of a sublease agreement. The function accepts a text string and a list of details to extract as inputs. In this example, we call the function with the `document_text` and `details_to_extract` variables that were defined earlier.

Within the function, a prompt is generated for Claude, including the document to be summarized, the details to extract, and specific instructions for summarizing the document. The prompt instructs Claude to respond with a summary of each detail to extract nested within XML headers.

Because we decided to output each section of the summary within tags, each section can easily be parsed out as a post-processing step. This approach enables structured summaries that can be adapted for your use case, so that each summary follows the same pattern.

Evaluate your prompt

Prompting often requires testing and optimization for it to be production ready. To determine the readiness of your solution, evaluate the quality of your summaries using a systematic process combining quantitative and qualitative methods. Creating a [strong empirical evaluation](#) based on your defined success criteria will allow you to optimize your prompts. Possible metrics include:

- **ROUGE scores:** Overlap between generated and reference summaries, focusing on recall.
- **BLEU scores:** Precision of n-gram matches, useful for phrase similarity.
- **Contextual embedding similarity:** Semantic similarity using vector embeddings.
- **LLM-based grading:** Using Claude to evaluate summaries against a rubric.
- **Human evaluation:** Legal experts assess quality, especially at small scale.

Deploy your prompt

Additional considerations for deployment:

1. **Ensure no liability:** Include disclaimers clarifying AI-generated summaries should be reviewed by legal professionals.
2. **Handle diverse document types:** Ensure your pipeline can process PDFs, Word, text files, etc.
3. **Parallelize API calls:** For large collections, send requests concurrently, respecting [rate limits](#).

Improve performance

For complex cases, consider strategies beyond prompt engineering:

Perform meta-summarization to summarize long documents

Use chunking to process long docs exceeding Claude's context window:

```
import anthropic

client = anthropic.Anthropic()

def chunk_text(text, chunk_size=20000):
    return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]

def summarize_long_document(text, details_to_extract, model="claude-opus-4-20250514",
                             details_str = '\n'.join(details_to_extract))
    chunks = chunk_text(text)
    chunk_summaries = [summarize_document(chunk, details_to_extract, model, max_tokens

    final_prompt = f"""
```

You are looking at the chunked summaries of multiple documents that are all related. Combine the following summaries of the document from different truthful sources into a <chunked_summaries> {'.join(chunk_summaries)} </chunked_summaries>

Focus on these key aspects:

```
{details_str}
"""
```

```
response = client.messages.create(
    model=model,
    max_tokens=max_tokens,
    system="You are a legal expert that summarizes notes on one document.",
    messages=[
        {"role": "user", "content": final_prompt},
        {"role": "assistant", "content": "Here is the combined summary:"}
    ],
    stop_sequences=["</summary>"]
)

return response.content[0].text
```

This technique can help summarize documents beyond Claude's token limit.

Use summary indexed documents to explore

Generate summaries for each document in your corpus, then use Claude to rank relevance, improving retrieval efficiency.

Fine-tune Claude

Fine-tuning on your dataset can improve the accuracy of summaries:

- Collect examples of underperforming summaries.
- Compile a curated dataset of documents and corrected summaries.
- Retrain Claude via Amazon Bedrock.
- Continuously add feedback to refine.

More on fine-tuning is available in the [AWS launch blog](#).

Resources

- [Summarization cookbook](#)
- [Citations cookbook](#)

Feedback

Was this page helpful?

Yes No

External links

- [Content moderation](#)

- [Overview](#)

Connect

- [X](#)
- [LinkedIn](#)
- [Discord](#)

On this page

- [Before building with Claude](#)
- [Decide whether to use Claude for legal summarization](#)
- [Determine the details you want the summarization to extract](#)
- [Establish success criteria](#)
- [How to summarize legal documents using Claude](#)
- [Select the right Claude model](#)
- [Transform documents into a format Claude can process](#)
- [Build a strong prompt](#)
- [Evaluate your prompt](#)
- [Deploy your prompt](#)
- [Improve performance](#)
- [Perform meta-summarization](#)
- [Use summary indexed documents](#)
- [Fine-tune Claude](#)

Prompt engineering overview - Anthropic

Prompt engineering overview

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

Before prompt engineering

This guide assumes that you have:

1. A clear definition of the success criteria for your use case
2. Some ways to empirically test against those criteria
3. A first draft prompt you want to improve

If not, we highly suggest you spend time establishing that first. Check out [Define your success criteria](#) and [Create strong empirical evaluations](#) for tips and guidance.

 [Prompt generator](#)

When to prompt engineer

This guide focuses on success criteria that are controllable through prompt engineering. Not every success criterion or failing evaluation is best solved by prompt engineering. For example, latency and cost can sometimes be improved by selecting a different model.

Prompting vs. finetuning

Prompt engineering is far faster than other methods of model behavior control, such as finetuning, and can often yield leaps in performance in far less time. Here are some reasons to consider prompt engineering over finetuning:

- **Resource efficiency:** Fine-tuning requires high-end GPUs and large memory, while prompt engineering only needs text input, making it much more resource-friendly.
- **Cost-effectiveness:** For cloud-based AI services, fine-tuning incurs significant costs. Prompt engineering uses the base model, which is typically cheaper.
- **Maintaining model updates:** When providers update models, fine-tuned versions might need retraining. Prompts usually work across versions without changes.
- **Time-saving:** Fine-tuning can take hours or even days. In contrast, prompt engineering provides nearly instantaneous results, allowing for quick problem-solving.
- **Minimal data needs:** Fine-tuning needs substantial task-specific, labeled data, which can be scarce or expensive. Prompt engineering works with few-shot or even zero-shot learning.
- **Flexibility & rapid iteration:** Quickly try various approaches, tweak prompts, and see immediate results. This rapid experimentation is difficult with fine-tuning.
- **Domain adaptation:** Easily adapt models to new domains by providing domain-specific context in prompts, without retraining.
- **Comprehension improvements:** Prompt engineering is far more effective than finetuning at helping models better understand and utilize external content such as retrieved documents.
- **Preserves general knowledge:** Fine-tuning risks catastrophic forgetting, where the model loses general knowledge. Prompt engineering maintains the model's broad capabilities.

- **Transparency:** Prompts are human-readable, showing exactly what information the model receives. This transparency aids in understanding and debugging.
-

How to prompt engineer

The prompt engineering pages in this section have been organized from most broadly effective techniques to more specialized techniques. When troubleshooting performance, we suggest you try these techniques in order, although the actual impact of each technique will depend on your use case.

1. [Prompt generator](#)
 2. [Be clear and direct](#)
 3. [Use examples \(multishot\)](#)
 4. [Let Claude think \(chain of thought\)](#)
 5. [Use XML tags](#)
 6. [Give Claude a role \(system prompts\)](#)
 7. [Prefill Claude's response](#)
 8. [Chain complex prompts](#)
 9. [Long context tips](#)
-

Prompt engineering tutorial

If you're an interactive learner, you can dive into our interactive tutorials instead!

- [GitHub prompting tutorial](#)
An example-filled tutorial that covers the prompt engineering concepts found in our docs.
 - [Google Sheets prompting tutorial](#)
A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.
-

Feedback

Was this page helpful?

Yes No

[Legal summarization](#)

[Claude 4 best practices](#)

Follow us



[LinkedIn](#)

[Discord](#)

On this page

- [Before prompt engineering](#)
- [When to prompt engineer](#)
- [How to prompt engineer](#)
- [Prompt engineering tutorial](#)

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Just a moment...

Enable JavaScript and cookies to continue

Automatically generate first draft prompt templates

Our prompt generator is compatible with all Claude models, including those with extended thinking capabilities. For prompting tips specific to extended thinking models, see [here](#).

Sometimes, the hardest part of using an AI model is figuring out how to prompt it effectively. To help with this, we've created a prompt generation tool that guides Claude to generate high-quality prompt templates tailored to your specific tasks. These templates follow many of our prompt engineering best practices.

The prompt generator is particularly useful as a tool for solving the “blank page problem” to give you a jumping-off point for further testing and iteration.

Try the prompt generator now directly on the [Console](#).

If you're interested in analyzing the underlying prompt and architecture, check out our [prompt generator Google Colab notebook](#). There, you can easily run the code to have Claude construct prompts on your behalf.

Note that to run the Colab notebook, you will need an [API key](#).

Next steps

[Start prompt engineering](#)

Get inspired by a curated selection of prompts for various tasks and use cases.

[Prompt library](#)

Get inspired by a curated selection of prompts for various tasks and use cases.

[GitHub prompting tutorial](#)

An example-filled tutorial that covers the prompt engineering concepts found in our docs.

[Google Sheets prompting tutorial](#)

A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.

Was this page helpful?

Yes No

[Claude 4 best practices](#)

[Use prompt templates](#)

On this page

- [Next steps](#)

Use prompt templates and variables

When deploying an LLM-based application with Claude, your API calls will typically consist of two types of content:

- **Fixed content:** Static instructions or context that remain constant across multiple interactions
- **Variable content:** Dynamic elements that change with each request or conversation, such as:
 - User inputs
 - Retrieved content for Retrieval-Augmented Generation (RAG)
 - Conversation context such as user account history
 - System-generated data such as tool use results fed in from other independent calls to Claude

A **prompt template** combines these fixed and variable parts, using placeholders for the dynamic content. In the [Anthropic Console](#), these placeholders are denoted with `{{double brackets}}`, making them easily identifiable and allowing for quick testing of different values.

When to use prompt templates and variables

You should always use prompt templates and variables when you expect any part of your prompt to be repeated in another call to Claude (only via the API or the [Anthropic Console](#)). [claude.ai](#) currently does not support prompt templates or variables.

Prompt templates offer several benefits:

- **Consistency:** Ensure a consistent structure for your prompts across multiple interactions
- **Efficiency:** Easily swap out variable content without rewriting the entire prompt
- **Testability:** Quickly test different inputs and edge cases by changing only the variable portion
- **Scalability:** Simplify prompt management as your application grows in complexity
- **Version control:** Easily track changes to your prompt structure over time by keeping tabs only on the core part of your prompt, separate from dynamic inputs

The [Anthropic Console](#) heavily uses prompt templates and variables to support features such as:

- **Prompt generator:** Decides what variables your prompt needs and includes them in the template it outputs
 - **Prompt improver:** Takes your existing template, including all variables, and maintains them in the improved template
 - **Evaluation tool:** Allows you to easily test, scale, and track versions of your prompts by separating the variable and fixed portions
-

Example prompt template

Let's consider a simple application that translates English text to Spanish. The translated text would be variable since you would expect this text to change between users or calls to Claude. This translated text could be dynamically retrieved from databases or the user's input.

Thus, for your translation app, you might use this simple prompt template:

Translate this text from English to Spanish: `{{text}}`

Next steps

- [Generate a prompt](#): Learn about the prompt generator in the Anthropic Console and try your hand at getting Claude to generate a prompt for you.
 - [Apply XML tags](#): If you want to level up your prompt variable game, wrap them in XML tags.
 - [Anthropic Console](#): Check out the myriad prompt development tools available.
-

Was this page helpful?

- Yes
 - No
-

Additional Links

- [Prompt generator](#)
 - [Prompt improver](#)
-

Connect with us



On this page

- [When to use prompt templates and variables](#)
- [Example prompt template](#)
- [Next steps](#)

Use our prompt improver to optimize your prompts

Our prompt improver is compatible with all Claude models, including those with extended thinking capabilities. For prompting tips specific to extended thinking models, see [here](#).

The prompt improver helps you quickly iterate and improve your prompts through automated analysis and enhancement. It excels at making prompts more robust for complex tasks that require high accuracy.

Before you begin

You'll need:

- A [prompt template](#) to improve
- Feedback on current issues with Claude's outputs (optional but recommended)
- Example inputs and ideal outputs (optional but recommended)

How the prompt improver works

The prompt improver enhances your prompts in 4 steps:

1. **Example identification:** Locates and extracts examples from your prompt template
2. **Initial draft:** Creates a structured template with clear sections and XML tags
3. **Chain of thought refinement:** Adds and refines detailed reasoning instructions
4. **Example enhancement:** Updates examples to demonstrate the new reasoning process

You can watch these steps happen in real-time in the improvement modal.

What you get

The prompt improver generates templates with:

- Detailed chain-of-thought instructions that guide Claude's reasoning process and typically improve its performance
- Clear organization using XML tags to separate different components
- Standardized example formatting that demonstrates step-by-step reasoning from input to output
- Strategic prefills that guide Claude's initial responses

While examples appear separately in the Workbench UI, they're included at the start of the first user message in the actual API call. View the raw format by clicking `</> Get Code` or insert examples as raw text via the Examples box.

How to use the prompt improver

1. Submit your prompt template
2. Add any feedback about issues with Claude's current outputs (e.g., "summaries are too basic for expert audiences")
3. Include example inputs and ideal outputs
4. Review the improved prompt

Generate test examples

Don't have examples yet? Use our [Test Case Generator](#) to:

- Generate sample inputs
- Get Claude's responses
- Edit the responses to match your ideal outputs
- Add the polished examples to your prompt

When to use the prompt improver

The prompt improver works best for:

- Complex tasks requiring detailed reasoning
- Situations where accuracy is more important than speed
- Problems where Claude's current outputs need significant improvement

For latency or cost-sensitive applications, consider using simpler prompts. The prompt improver creates templates that produce longer, more thorough, but slower responses.

Example improvement

Here's how the prompt improver enhances a basic classification prompt:

Original prompt

From the following list of Wikipedia article titles, identify which article this sentence belongs to. Respond with just the article title and nothing else.

Article titles:

```
{{titles}}
```

Sentence to classify:

```
{{sentence}}
```

Improved prompt

You are an intelligent text classification system specialized in matching sentences to

First, review the following list of Wikipedia article titles:

```
<article_titles>
```

```
{{titles}}
```

```
</article_titles>
```

Now, consider this sentence that needs to be classified:

```
<sentence_to_classify>
```

```
{{sentence}}
```

```
</sentence_to_classify>
```

Your goal is to determine which article title from the provided list best matches the

1. List the key concepts from the sentence
2. Compare each key concept with the article titles
3. Rank the top 3 most relevant titles and explain why they are relevant
4. Select the most appropriate article title that best encompasses or relates to the s

Wrap your analysis in <analysis> tags. Include the following:

- List of key concepts from the sentence
- Comparison of each key concept with the article titles
- Ranking of top 3 most relevant titles with explanations
- Your final choice and reasoning

After your analysis, provide your final answer: the single most appropriate Wikipedia

Output only the chosen article title, without any additional text or explanation.

Notice how the improved prompt:

- Adds clear step-by-step reasoning instructions
- Uses XML tags to organize content
- Provides explicit output formatting requirements
- Guides Claude through the analysis process

Troubleshooting

Common issues and solutions:

- **Examples not appearing in output:** Check that examples are properly formatted with XML tags and appear at the start of the first user message
 - **Chain of thought too verbose:** Add specific instructions about desired output length and level of detail
 - **Reasoning steps don't match your needs:** Modify the steps section to match your specific use case
-

Next steps

- [Prompt library](#): Get inspired by example prompts for various tasks.
 - [GitHub prompting tutorial](#): Learn prompting best practices with our interactive tutorial.
 - [Test your prompts](#): Use our evaluation tool to test your improved prompts.
-

Was this page helpful?

Yes / No

[Use prompt templates](#) | [Be clear and direct](#)

Follow us on [X](#), [LinkedIn](#), [Discord](#).

Be clear, direct, and detailed - Anthropic

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

When interacting with Claude, think of it as a brilliant but very new employee (with amnesia) who needs explicit instructions. Like any new employee, Claude does not have context on your norms, styles, guidelines, or preferred ways of working.

The more precisely you explain what you want, the better Claude's response will be.

The golden rule of clear prompting

Show your prompt to a colleague, ideally someone who has minimal context on the task, and ask them to follow the instructions. If they're confused, Claude will likely be too.

How to be clear, contextual, and specific

- **Give Claude contextual information:**

Just like you might be able to better perform on a task if you knew more context, Claude will perform better if it has more contextual information. Some examples include:

- What the task results will be used for
- What audience the output is meant for
- What workflow the task is a part of, and where this task belongs in that workflow
- The end goal of the task, or what a successful task completion looks like

- **Be specific about what you want Claude to do:**

For example, if you want Claude to output only code and nothing else, say so.

- **Provide instructions as sequential steps:**

Use numbered lists or bullet points to better ensure that Claude carries out the task exactly as you want.

Examples

Example: Anonymizing customer feedback

Notice that Claude still makes mistakes in the unclear prompting example, such as leaving in a customer's name.

Role	Unclear Prompt	Clear Prompt
User	Please remove all personally identifiable information from these customer feedback messages: <code>{{ FEEDBACK_DATA }}</code>	Your task is to anonymize customer feedback for our quarterly review.

Instructions:

1. Replace all customer names with "CUSTOMER_[ID]" (e.g., "Jane Doe" → "CUSTOMER_001").
2. Replace email addresses with "EMAIL_[ID]@example.com".

3. Redact phone numbers as "PHONE_[ID]".
4. If a message mentions a specific product (e.g., "AcmeCloud"), leave it intact.
5. If no PII is found, copy the message verbatim.
6. Output only the processed messages, separated by `---`.

Data to process: `{{FEEDBACK_DATA}}` |

Assistant (Claude's response)

Here are the anonymized customer feedback messages:

1. I've been using your software for years, and it's been great. But lately, AcmeCloud has been crashing a lot. Fix it!
2. XXX, your customer support rep, was incredibly helpful. He guided me through the whole process. My email is XXX if you want to send a survey.
3. I'm John Smith, and I'm not happy. I've been on hold for 2 hours trying to reach someone at XXX. This is unacceptable! |
 || `CUSTOMER_001` : I've been using your software for years, and it's been great. But lately, AcmeCloud has been crashing a lot. Fix it!

CUSTOMER_002 : CUSTOMER_SERVICE_001 , your customer support rep, was incredibly helpful. He guided me through the whole process. My email is EMAIL_002@example.com if you want to send a survey.

`CUSTOMER_003` : I'm `CUSTOMER_003` , and I'm not happy. I've been on hold for 2 hours trying to reach someone at `PHONE_001` . This is unacceptable! |

Example: Crafting a marketing email campaign

Notice that Claude makes up details to fill in the gaps where it lacks context with the vague prompt.

Role	Vague Prompt	Specific Prompt
User	Write a marketing email for our new AcmeCloud features.	Your task is to craft a targeted marketing email for our Q3 AcmeCloud feature release.

Instructions:

1. Write for this target audience: Mid-size tech companies (100-500 employees) upgrading from on-prem to cloud.
2. Highlight 3 key new features: advanced data encryption, cross-platform sync, and real-time collaboration.
3. Tone: Professional yet approachable. Emphasize security, efficiency, and teamwork.
4. Include a clear CTA: Free 30-day trial with priority onboarding.
5. Subject line: Under 50 chars, mention "security" and "collaboration".
6. Personalization: Use `{{COMPANY_NAME}}` and `{{CONTACT_NAME}}` variables.
 Structure:
 7. Subject line
 8. Email body (150-200 words)
 9. CTA button text |

Assistant (Claude's response)

Here's a draft marketing email for our new features:

Subject: Boost Your Business with AcmeCloud's Latest Features

Dear Valued Customer,

We're excited to introduce three powerful features to AcmeCloud that will revolutionize your workflow:

1. Real-Time Collaboration: Work on documents simultaneously with your team. No more version control issues!
2. AI-Powered Insights: Make data-driven decisions with AI that analyzes trends and automates tasks.
3. Advanced Security Suite: End-to-end encryption, multi-factor auth, and threat detection keep your data safe.

Limited Offer: Upgrade now for a 20% first-year discount. Plus, refer a friend for an extra 10% off!

Best,
The AcmeCloud Team |
|| Subject line: AcmeCloud: Enterprise security meets real-time teamwork

Email body:

Dear `{{CONTACT_NAME}}`,

At `{{COMPANY_NAME}}`, you're scaling fast and eyeing the cloud. But with growth comes the challenge of keeping data secure while fostering collaboration. That's where AcmeCloud's Q3 release comes in, tailor-made for tech innovators like you.

1. Fort Knox for Your Files: Military-grade encryption now safeguards your data, both in motion and at rest. Your intellectual property stays yours, period.
2. Sync Without Borders: Windows, Mac, or Linux—AcmeCloud speaks all languages. Unified version control means no more “final_v2_REALLY_FINAL.doc”.
3. Collaborate at Light Speed: Ditch the attachment ping-pong. Now your devs, designers, and data scientists co-create in real-time, slashing time-to-market.

Don't just take our word for it. Experience the AcmeCloud difference with our 30-day trial. And because we know time is your most precious asset, we're throwing in priority onboarding.

Elevate Your Team Now - Free 30-Day Trial |

Example: Incident response

Notice that Claude outputs superfluous text and different formatting with the vague prompt.

Role	Vague Prompt	Detailed Prompt
User	Analyze this AcmeCloud outage report and summarize the key points. <code>{{REPORT}}</code>	Analyze this AcmeCloud outage report. Skip the preamble. Keep your response terse and write only the bare bones necessary information. List only:
	<ol style="list-style-type: none">1. Cause2. Duration3. Impacted services4. Number of affected users5. Estimated revenue loss.	
	Here's the report: <code>{{REPORT}}</code>	

Assistant (Claude's response)

Here's a summary of the key points from the AcmeCloud outage report:

On May 15th, a misconfigured load balancer update caused significant traffic routing issues, resulting in a service outage from 09:15 to 13:52 UTC. The 4-hour 37-minute disruption impacted our core file syncing service and admin dashboard, rendering them inaccessible. Additionally, the mobile app experienced partial functionality loss. Approximately 68% of our user base, totaling 1.3 million users, were affected. Based on our Service Level Agreements (SLAs) and average revenue per user, we estimate a financial impact of \$420,000 in service credits and potential customer churn.

1) Cause:
Misconfigured load balancer update

2) Duration: 4h 37m (09:15-13:52 UTC, May 15)

3) Impacted: Core sync, admin dashboard (down); mobile app (partial)

4) Affected users: 1.3M (68% of base)

5) Est. revenue loss: \$420,000

Additional Resources

[Prompt library](#)

Get inspired by a curated selection of prompts for various tasks and use cases.

[GitHub prompting tutorial](#)

An example-filled tutorial that covers the prompt engineering concepts found in our docs.

[Google Sheets prompting tutorial](#)

A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.

Feedback

Was this page helpful?

Yes No

Navigational links

- [How to be clear, contextual, and specific](#)
 - [Examples](#)
-

Social links

- [X](#)
- [LinkedIn](#)
- [Discord](#)

Use examples (multishot prompting) to guide Claude's behavior

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

Examples are your secret weapon shortcut for getting Claude to generate exactly what you need. By providing a few well-crafted examples in your prompt, you can dramatically improve the accuracy, consistency, and quality of Claude's outputs.

This technique, known as few-shot or multishot prompting, is particularly effective for tasks that require structured outputs or adherence to specific formats.

Power up your prompts: Include 3-5 diverse, relevant examples to show Claude exactly what you want. More examples = better performance, especially for complex tasks.

Why use examples?

- **Accuracy:** Examples reduce misinterpretation of instructions.
- **Consistency:** Examples enforce uniform structure and style.
- **Performance:** Well-chosen examples boost Claude's ability to handle complex tasks.

Crafting effective examples

For maximum effectiveness, make sure that your examples are:

- **Relevant:** Your examples mirror your actual use case.
- **Diverse:** Your examples cover edge cases and potential challenges, and vary enough that Claude doesn't inadvertently pick up on unintended patterns.
- **Clear:** Your examples are wrapped in `<example>` tags (if multiple, nested within `<examples>` tags) for structure.

Ask Claude to evaluate your examples for relevance, diversity, or clarity. Or have Claude generate more examples based on your initial set.

Example: Analyzing customer feedback

Notice that in the "no examples" version, Claude does not list multiple options for each category, despite its written analysis indicating that there are multiple categories for certain feedback. It also includes a long explanation when we may not want it to.

Role	No Examples	With Examples
User	Analyze this customer feedback and categorize the issues. Use these categories: UI/UX, Performance, Feature Request, Integration, Pricing, and Other. Also rate the sentiment (Positive/Neutral/Negative) and priority (High/Medium/Low).	
Here is the feedback: <code>{{FEEDBACK}}</code>	Our CS team is overwhelmed with unstructured feedback. Your task is to analyze feedback and categorize issues for our product and engineering teams. Use these categories: UI/UX, Performance, Feature Request, Integration, Pricing, and Other. Also rate the sentiment (Positive/Neutral/Negative) and priority (High/Medium/Low). Here is an example:	

Input: The new dashboard is a mess! It takes forever to load, and I can't find the export button. Fix this ASAP!
Category: UI/UX, Performance Sentiment: Negative Priority: High

Now, analyze this feedback: {{FEEDBACK}} |

- Here's the analysis of the customer feedback:
 1. The new dashboard is a mess! It takes forever to load, and I can't find the export button. Fix this ASAP!
Category: UI/UX, Performance
Sentiment: Negative
Priority: High
 2. Love the Salesforce integration! But it'd be great if you could add Hubspot too.
Category: Integration, Feature Request
Sentiment: Positive
Priority: Medium
 3. Your pricing is ridiculous! \$99/mo for basic features? I'm switching to [Competitor X](#).
Category: Pricing
Sentiment: Negative
Priority: High
 4. The mobile app keeps crashing when I try to upload large files (>50MB). This is critical for our field agents.
Category: Performance
Sentiment: Negative
Priority: High
 5. Overall, I'm happy with the product. The UI is intuitive, and the customer support is top-notch. Keep it up!
Category: UI/UX, Other (Customer Support)
Sentiment: Positive
Priority: Low |

Prompt library

Get inspired by a curated selection of prompts for various tasks and use cases.

[Explore the prompt library](#)

[GitHub prompting tutorial](#)

An example-filled tutorial that covers the prompt engineering concepts found in our docs.

[Google Sheets prompting tutorial](#)

A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.

Was this page helpful?

[Yes](#) | [No](#)

Helpful links:

[Be clear and direct](#)

[Let Claude think \(CoT\)](#)

Connect with us

Let Claude think (chain of thought prompting) to increase performance

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

When faced with complex tasks like research, analysis, or problem-solving, giving Claude space to think can dramatically improve its performance. This technique, known as chain of thought (CoT) prompting, encourages Claude to break down problems step-by-step, leading to more accurate and nuanced outputs.

Before implementing CoT

Why let Claude think?

- **Accuracy:** Stepping through problems reduces errors, especially in math, logic, analysis, or generally complex tasks.
- **Coherence:** Structured thinking leads to more cohesive, well-organized responses.
- **Debugging:** Seeing Claude's thought process helps you pinpoint where prompts may be unclear.

Why not let Claude think?

- Increased output length may impact latency.
- Not all tasks require in-depth thinking. Use CoT judiciously to ensure the right balance of performance and latency.

Use CoT for tasks that a human would need to think through, like complex math, multi-step analysis, writing complex documents, or decisions with many factors.

How to prompt for thinking

The chain of thought techniques below are **ordered from least to most complex**. Less complex methods take up less space in the context window, but are also generally less powerful.

CoT tip: Always have Claude output its thinking. Without outputting its thought process, no thinking occurs!

Basic prompt

Include "Think step-by-step" in your prompt.

- Lacks guidance on *how* to think (which is especially not ideal if a task is very specific to your app, use case, or organization)

Example: Writing donor emails (basic CoT)

Role: User

Content: Draft personalized emails to donors asking for contributions to this year's C

Program information:
<program>{{PROGRAM_DETAILS}}</program>

Donor information:
<donor>{{DONOR_DETAILS}}</donor>

Think step-by-step before you write the email.

Guided prompt

Outline specific steps for Claude to follow in its thinking process.

- Lacks structuring to make it easy to strip out and separate the answer from the thinking.

Example: Writing donor emails (guided CoT)

Role: User

Content: Draft personalized emails to donors asking for contributions to this year's C

Program information:
<program>{{PROGRAM_DETAILS}}</program>

Donor information:
<donor>{{DONOR_DETAILS}}</donor>

Think before you write the email. First, think through what messaging might appeal to

Structured prompt

Use XML tags like `<thinking>` and `<answer>` to separate reasoning from the final answer.

Example: Writing donor emails (structured guided CoT)

Role: User

Content: Draft personalized emails to donors asking for contributions to this year's C

Program information:
<program>{{PROGRAM_DETAILS}}</program>

Donor information:
<donor>{{DONOR_DETAILS}}</donor>

Think before you write the email in `<thinking>` tags. First, think through what messagi

Examples

Financial analysis without thinking

Without step-by-step thinking, Claude gives a recommendation that seems reasonable but lacks depth. It doesn't quantify the potential outcomes or consider historical market performance, which are crucial for making an informed decision.

Role	Content
User	You're a financial advisor. A client wants to invest \$10,000. They can choose between two options: A) A stock that historically returns 12% annually but is volatile, or B) A bond that guarantees 6% annually. The client needs the money in 5 years for a down payment on a house. Which option do you recommend?
Assistant	Based on the information provided, I would recommend option B, the bond that guarantees a 6% annual return. Here's why: ...

Financial analysis with thinking

With step-by-step thinking, the advisor provides a much richer analysis. It calculates exact figures for both scenarios, considers historical market volatility, and deeply analyzes the client's risk tolerance. This thorough reasoning leads to a more confident and justifiable recommendation.

Role	Content
User	You're a financial advisor. A client wants to invest \$10,000. They can choose between two options: A) A stock that historically returns 12% annually but is volatile, or B) A bond that guarantees 6% annually. The client needs the money in 5 years for a down payment on a house. Which option do you recommend? Think step-by-step.
Assistant	To determine the best investment option for our client, I'll consider several factors step-by-step: ...

Additional links and resources

- [Prompt library](#): Get inspired by a curated selection of prompts for various tasks and use cases.
- [GitHub prompting tutorial](#): An example-filled tutorial that covers the prompt engineering concepts found in our docs.
- [Google Sheets prompting tutorial](#): A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.

Was this page helpful?

Yes | No

Follow these topics for more:

- [Use examples \(multishot prompting\)](#)
- [Use XML tags](#)

Use XML tags to structure your prompts

Copy page

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

When your prompts involve multiple components like context, instructions, and examples, XML tags can be a game-changer. They help Claude parse your prompts more accurately, leading to higher-quality outputs.

XML tip: Use tags like `<instructions>`, `<example>`, and `<formatting>` to clearly separate different parts of your prompt. This prevents Claude from mixing up instructions with examples or context.

Why use XML tags?

- **Clarity:** Clearly separate different parts of your prompt and ensure your prompt is well structured.
- **Accuracy:** Reduce errors caused by Claude misinterpreting parts of your prompt.
- **Flexibility:** Easily find, add, remove, or modify parts of your prompt without rewriting everything.
- **Parseability:** Having Claude use XML tags in its output makes it easier to extract specific parts of its response by post-processing.

There are no canonical “best” XML tags that Claude has been trained with in particular, although we recommend that your tag names make sense with the information they surround.

Tagging best practices

1. **Be consistent:** Use the same tag names throughout your prompts, and refer to those tag names when talking about the content (e.g, `<contract>`).
2. **Nest tags:** You should nest tags `<outer><inner></inner></outer>` for hierarchical content.

Power user tip: Combine XML tags with other techniques like multishot prompting (`<examples>`) or chain of thought (`<thinking>` , `<answer>`). This creates super-structured, high-performance prompts.

Examples

Example: Generating financial reports

Without XML tags, Claude misunderstands the task and generates a report that doesn't match the required structure or tone. After substitution, there is also a chance that Claude misunderstand where one section (like the the Q1 report example) stops and another begins.

Role	No XML Tags	With XML Tags
User	You're a financial analyst at AcmeCorp. Generate a Q2 financial report for our investors. Include sections on Revenue Growth, Profit Margins, and Cash Flow, like with this example from last	You're a financial analyst at AcmeCorp. Generate a Q2 financial report for our

Role	No XML Tags	With XML Tags
	<p>year: {{Q1_REPORT}}. Use data points from this spreadsheet: {{SPREADSHEET_DATA}}. The report should be extremely concise, to the point, professional, and in list format. It should and highlight both strengths and areas for improvement.</p>	<p>investors.</p> <p>AcmeCorp is a B2B SaaS company. Our investors value transparency and actionable insights.</p> <p>Use this data for your report: <data>{{SPREADSHEET_DATA}}</data></p> <p><instructions> 1. Include sections: Revenue Growth, Profit Margins, Cash Flow. 2. Highlight strengths and areas for improvement. </instructions></p> <p>Make your tone concise and professional. Follow this structure: <formatting_example> {{Q1_REPORT}} </formatting_example></p>

Assistant Q2 Financial Report for AcmeCorp Investors

Dear Valued Investors,

I am pleased to present our Q2 financial report, showcasing AcmeCorp's continued growth and resilience in the B2B SaaS market.

1. Revenue Growth:

Our Q2 revenue reached \$15.2M, a 22% YoY increase. This growth is primarily driven by a 30% expansion in our enterprise client base. However, our SMB segment saw a slight decline of 5%, which we aim to address through targeted marketing initiatives in Q3.

2. Profit Margins:

Gross margin improved to 72% (up 3% from Q1) due to infrastructure optimizations. EBITDA margin, however, dipped to 18% (down 2%) as we invested heavily in R&D for our upcoming AI-driven features.

3. Cash Flow:

Operating cash flow remains strong at \$4.1M. Our cash reserves stand at \$28M, providing a 15-month runway at current burn rate. We did see a 20% increase in customer acquisition costs, which we're addressing through more efficient sales strategies.

In summary, AcmeCorp's core financials remain robust. We're balancing growth investments with prudent cash management. Our focus for Q3 will be revitalizing SMB growth and optimizing sales efficiency.

Role	No XML Tags	With XML Tags
	Thank you for your continued trust.	
	Sincerely, CFO, AcmeCorp	

Example: Legal contract analysis

Without XML tags, Claude’s analysis is disorganized and misses key points. With tags, it provides a structured, thorough analysis that a legal team can act on.

Role	No XML Tags	With XML Tags
		Analyze this software licensing agreement for legal risks and liabilities.
		We’re a multinational enterprise considering this agreement for our core data infrastructure.
		<agreement> {{CONTRACT}}</agreement>
		This is our standard contract for reference:
	Analyze this software licensing agreement for potential risks and liabilities: {{CONTRACT}}. Focus on indemnification, limitation of liability, and IP ownership clauses. Also, note any unusual or concerning terms. Here’s our standard contract for reference: {{STANDARD_CONTRACT}}. Give a summary of findings and recommendations for our legal team.	<standard_contract> {{STANDARD_CONTRACT}} </standard_contract>
User		<instructions> 1. Analyze these clauses: - Indemnification - Limitation of liability - IP ownership 2. Note unusual or concerning terms. 3. Compare to our standard contract. 4. Summarize findings in <findings> tags. 5. List actionable recommendations in <recommendations> tags. </instructions>
Assistant	The provided software licensing agreement presents several concerning aspects. The indemnification clause is overly broad, potentially exposing us to significant liabilities. The limitation of liability section caps damages at a mere \$500, which is inadequate for our enterprise-level usage. Regarding IP, the agreement grants the vendor joint ownership of any modifications or integrations we create.	<findings> 1. Indemnification (Clause 8): - Issue: Overly broad. We indemnify vendor against “any and all claims” arising from our use. - Risk: Could hold us liable even for vendor’s negligence or product defects. - Deviation: Our standard limits indemnity to claims arising from our violation of agreement.
	In summary, this agreement is not in our best interest. It exposes us to undue risk, limits our recourse in case of failures, and compromises our intellectual property rights. I recommend renegotiating these terms or seeking alternative vendors.	2. Limitation of Liability (Clause

Role

No XML Tags

With XML Tags

9):

- Issue: Caps all damages at \$500, including direct damages.

- Risk: Grossly inadequate for enterprise use. A critical failure could cost millions.

- Deviation: Our standard allows for direct damages up to 12 months of fees paid.

3. IP Ownership (Clause 12):

- Issue: Grants vendor joint ownership of our modifications/integrations.

- Risk: Vendor could use or sell our proprietary algorithms or data insights.

- Deviation: Our standard retains full ownership of our work.

4. Unusual Terms:

- Clause 5.3: Vendor can use our usage data for "any purpose."

- Clause 7.2: Vendor can terminate for "convenience" with only 7 days' notice.

- Clause 11: No warranties of any kind, even for basic merchantability.

</findings>

<recommendations>

1. Reject this agreement. Risks far outweigh benefits for an enterprise.

2. Counter-propose:

- Indemnification: Limit to claims arising from our violation of agreement.

- Liability: Cap at 12 months' fees for direct damages; negotiate for indirect.

- IP: Full ownership of our modifications/integrations.

- Data Use: Restrict to service improvement only.

- Termination: Minimum 90 days' notice; only for cause.

- Warranties: Demand basic merchantability and fitness for purpose.

3. If vendor refuses, explore alternative providers or in-house solutions.

4. Engage legal counsel for contract negotiation given high

Role

No XML Tags

With XML Tags

stakes.

</recommendations>

Giving Claude a role with a system prompt

While these tips apply broadly to all Claude models, you can find prompting tips specific to extended thinking models [here](#).

When using Claude, you can dramatically improve its performance by using the `system` parameter to give it a role. This technique, known as role prompting, is the most powerful way to use system prompts with Claude.

The right role can turn Claude from a general assistant into your virtual domain expert!

System prompt tips: Use the `system` parameter to set Claude's role. Put everything else, like task-specific instructions, in the `user` turn instead.

Why use role prompting?

- **Enhanced accuracy:** In complex scenarios like legal analysis or financial modeling, role prompting can significantly boost Claude's performance.
- **Tailored tone:** Whether you need a CFO's brevity or a copywriter's flair, role prompting adjusts Claude's communication style.
- **Improved focus:** By setting the role context, Claude stays more within the bounds of your task's specific requirements.

How to give Claude a role

Use the [Messages API](#) to set Claude's role:

```
import anthropic

client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-3-7-sonnet-20250219",
    max_tokens=2048,
    system="You are a seasoned data scientist at a Fortune 500 company.", # <--- role
    messages=[
        {"role": "user", "content": "Analyze this dataset for anomalies: <dataset>{{DA
    ]
)

print(response.content)
```

Role prompting tip: Experiment with roles! A `data scientist` might see different insights than a `marketing strategist` for the same data. A `data scientist specializing in customer insight analysis for Fortune 500 companies` might yield different results still!

Examples

Example 1: Legal contract analysis

Without a role, Claude might miss critical issues:

Role	Content
User	Analyze this software licensing agreement for potential risks: <contract> {{CONTRACT}} </contract>
Assistant	Focus on indemnification, liability, and IP ownership. Here's a summary of the key points in the software licensing agreement: 1. Indemnification (Section 8): - We indemnify the vendor against claims arising from our use. 2. Limitation of Liability (Section 9): - Damages capped at \$500. 3. IP Ownership (Section 12): - We retain ownership of our modifications. Overall, the agreement seems standard. The indemnification and liability clauses are typical, and we maintain our IP rights.

With a role, Claude catches critical issues that could cost millions:

Role	Content
User	You are the General Counsel of a Fortune 500 tech company. We're considering this software licensing agreement for our core data infrastructure: <contract> {{CONTRACT}} </contract>
Assistant	Analyze it for potential risks, focusing on indemnification, liability, and IP ownership. Give your professional opinion. I've reviewed the software licensing agreement, and I have serious concerns that could expose our company to significant risks: 1. Indemnification (Section 8): - Critical Issue: We indemnify the vendor against "any and all claims" arising from our use. - Risk: This could hold us liable even for the vendor's negligence or product defects. - Recommendation: Reject. Limit our indemnity to claims arising from our violation of the agreement. 2. Limitation of Liability (Section 9): - Critical Issue: Damages capped at \$500, including direct damages. - Risk: Grossly inadequate. A critical failure in our data infrastructure could cost us millions in lost revenue and reputational damage. - Recommendation: Reject. Negotiate for cap at 12 months of fees for direct damages, and carve-outs for data breaches. 3. IP Ownership (Section 12): - Critical Issue: Vendor gains joint ownership of our modifications and integrations. - Risk: They could use or sell our proprietary algorithms or data insights to competitors. - Recommendation: Reject. We must retain full ownership of our work. Professional Opinion: Do not sign this agreement. It is grossly unfavorable and exposes us to unacceptable levels of risk. If the vendor refuses to renegotiate these terms, I strongly advise

Role	Content
	exploring alternative providers or accelerating our in-house solution. The potential costs of proceeding with this agreement far outweigh any short-term gains.

Example 2: Financial analysis

Without a role, Claude's analysis lacks depth:

Role	Content
User	Analyze this dataset of our Q2 financials: <data> {{FINANCIALS}} </data>
Assistant	<p>Highlight key trends and recommend actions. Here's a summary of your Q2 financials:</p> <ol style="list-style-type: none"> 1. Revenue: \$15.2M, up 22% YoY. Enterprise client base grew 30%. 2. Gross Margin: 72%, up 3% from Q1. 3. EBITDA Margin: 18%, down 2% due to R&D investments. 4. Cash Flow: \$4.1M operating, \$28M in reserves. <p>Overall, financials look strong with revenue growth and healthy margins. Increased R&D spending is impacting profitability but could pay off long-term. Maintain current course.</p>

With a role, Claude delivers actionable insights:

Role	Content
User	You are the CFO of a high-growth B2B SaaS company. We're in a board meeting discussing our Q2 financials: <data> {{FINANCIALS}} </data>
Assistant	<p>Analyze key trends, flag concerns, and recommend strategic actions. Our investors want aggressive growth but are wary of our burn rate.</p> <p>Board members, here's my analysis of our Q2 financials:</p> <ol style="list-style-type: none"> 1. Revenue (\$15.2M, +22% YoY): <ul style="list-style-type: none"> - Insight: Enterprise growth (+30%) is outpacing overall growth. - Flag: SMB segment declined 5%. - Action: Reallocate 20% of SMB marketing budget to enterprise sales. Task product to simplify SMB onboarding. 2. Margins (Gross: 72%, +3% QoQ; EBITDA: 18%, -2% QoQ): <ul style="list-style-type: none"> - Insight: Infrastructure optimizations boosted gross margins. - Flag: R&D spend on AI features dented EBITDA. - Action: No change. Our AI roadmap is critical for enterprise stickiness. 3. Cash (\$4.1M OCF, \$28M reserves): <ul style="list-style-type: none"> - Insight: 15-month runway at current burn. - Flag: CAC up 20%, eroding LTV/CAC. - Action: Freeze non-essential hires. Implement sales funnel analytics to cut CAC by 15% in Q3. <p>Strategic Stance: We're threading the needle between growth and burn. Enterprise focus is our growth engine—invest there. But watch SMB churn; it could bleed into enterprise. Our AI bet is sound, but we must convert it to ARR by Q4, or we cut R&D by 30%.</p>

Role

Content

In sum: Double down on enterprise, streamline SMB, optimize sales, and monetize AI. Questions?

Additional resources

- [Prompt library](#): Get inspired by a curated selection of prompts for various tasks and use cases.
 - [GitHub prompting tutorial](#): An example-filled tutorial that covers the prompt engineering concepts found in our docs.
 - [Google Sheets prompting tutorial](#): A lighter weight version of our prompt engineering tutorial via an interactive spreadsheet.
-

Footnotes

Was this page helpful?

Yes No

[Use XML tags](#)

[Prefill Claude's response](#)

On this page

- [Why use role prompting?](#)
 - [How to give Claude a role](#)
 - [Examples](#)
 - [Example 1: Legal contract analysis](#)
 - [Example 2: Financial analysis](#)
-

Follow Anthropic on [X](#), [LinkedIn](#), and [Discord](#).

Just a moment...

Enable JavaScript and cookies to continue

Create strong empirical evaluations - Anthropic

Introduction

Test & evaluate

Create strong empirical evaluations

Building evals and test cases

Eval design principles

1. **Be task-specific:** Design evals that mirror your real-world task distribution. Don't forget to factor in edge cases!
 - Example edge cases:
 - Irrelevant or nonexistent input data
 - Overly long input data or user input
 - [Chat use cases] Poor, harmful, or irrelevant user input
 - Ambiguous test cases where even humans would find it hard to reach an assessment consensus
2. **Automate when possible:** Structure questions to allow for automated grading (e.g., multiple-choice, string match, code-graded, LLM-graded).
3. **Prioritize volume over quality:** More questions with slightly lower signal automated grading is better than fewer questions with high-quality human hand-graded evals.

Example evals

Task fidelity (sentiment analysis) - exact match evaluation

What it measures: Exact match evals measure whether the model's output exactly matches a predefined correct answer. It's a simple, unambiguous metric suitable for tasks like sentiment analysis (positive, negative, neutral).

Example eval test cases: 1000 tweets with human-labeled sentiments.

```
import anthropic

tweets = [
    {"text": "This movie was a total waste of time. 🙄", "sentiment": "negative"},
    {"text": "The new album is 🔥! Been on repeat all day.", "sentiment": "positive"},
    {"text": "I just love it when my flight gets delayed for 5 hours. #bestdayever", "sentiment": "positive"},
    {"text": "The movie's plot was terrible, but the acting was phenomenal.", "sentiment": "mixed"},
    # ... 996 more tweets
]

client = anthropic.Anthropic()
```

```

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=50,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

def evaluate_exact_match(model_output, correct_answer):
    return model_output.strip().lower() == correct_answer.lower()

outputs = [get_completion(f"Classify this as 'positive', 'negative', 'neutral', or 'mixed'") for output, tweet in zip(
accuracy = sum(evaluate_exact_match(output, tweet['sentiment'])) for output, tweet in zip(
print(f"Sentiment Analysis Accuracy: {accuracy * 100}%")

```

Consistency (FAQ bot) - cosine similarity evaluation

What it measures: Cosine similarity measures the similarity between two vectors (sentence embeddings of outputs using SBERT), with values closer to 1 indicating higher similarity. It's ideal for evaluating consistency, as similar questions should yield semantically similar answers.

Example eval test cases: 50 groups with paraphrased versions each.

```

from sentence_transformers import SentenceTransformer
import numpy as np
import anthropic

faq_variations = [
    {"questions": ["What's your return policy?", "How can I return an item?", "Wut's y
# ... more FAQ groups
]

client = anthropic.Anthropic()

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=2048,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

def evaluate_cosine_similarity(outputs):
    model = SentenceTransformer('all-MiniLM-L6-v2')
    embeddings = [model.encode(output) for output in outputs]
    cosine_similarities = np.dot(embeddings, np.transpose(embeddings)) / (np.linalg.no
return np.mean(np.diag(cosine_similarities))

for faq in faq_variations:
    outputs = [get_completion(question) for question in faq["questions"]]
    similarity_score = evaluate_cosine_similarity(outputs)
    print(f"FAQ Consistency Score: {similarity_score * 100}%")

```

Relevance and coherence (summarization) - ROUGE-L evaluation

What it measures: ROUGE-L evaluates the quality of generated summaries by measuring the longest common subsequence (LCS) between the candidate and reference summaries. High scores indicate the summary captures key information coherently.

Example eval test cases: 200 articles with reference summaries.

```
from rouge import Rouge
import anthropic

articles = [
    {"text": "In a groundbreaking study, researchers at MIT...", "summary": "MIT scien
    {"text": "Jane Doe, a local hero, made headlines last week for saving... In city h
    {"text": "You won't believe what this celebrity did! ... extensive charity work ..
    # ... 197 more articles
]

client = anthropic.Anthropic()

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=1024,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

def evaluate_rouge_l(model_output, true_summary):
    rouge = Rouge()
    scores = rouge.get_scores(model_output, true_summary)
    return scores[0]['rouge-l']['f1'] # ROUGE-L F1 score

outputs = [get_completion(f"Summarize this article in 1-2 sentences:\n\n{article['text']}
relevance_scores = [evaluate_rouge_l(output, article['summary']) for output, article i
print(f"Average ROUGE-L F1 Score: {sum(relevance_scores)/len(relevance_scores)}")
```

Tone and style (customer service) - LLM-based Likert scale

What it measures: Uses an LLM to judge subjective attitudes like empathy, professionalism, or patience on a scale from 1 to 5.

Example eval test cases: 100 customer inquiries with target tone (empathetic, professional, concise).

```
import anthropic

inquiries = [
    {"text": "This is the third time you've messed up my order. I want a refund NOW! ",
    {"text": "I tried resetting my password but then my account got locked...", "tone"
    {"text": "I can't believe how good your product is. It's ruined all others for me!
```

```

    # ... 97 more inquiries
]

client = anthropic.Anthropic()

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=2048,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

def evaluate_likert(model_output, target_tone):
    tone_prompt = f"Rate this customer service response on a scale of 1-5 for being {t
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=50,
    messages=[{"role": "user", "content": tone_prompt}]
)
    return int(response.content[0].text.strip())

outputs = [get_completion(f"Respond to this customer inquiry: {inquiry['text']}") for
tone_scores = [evaluate_likert(output, inquiry['tone']) for output, inquiry in zip(out
print(f"Average Tone Score: {sum(tone_scores)/len(tone_scores)}")

```

Privacy preservation (medical chatbot) - LLM-based binary classification

What it measures: Classifies whether a response contains PHI (Personal Health Information). It can detect even implicit PHI missed by rule-based systems.

Example eval test cases: 500 simulated patient queries, some with PHI.

```

import anthropic

patient_queries = [
    {"query": "What are the side effects of Lisinopril?", "contains_phi": False},
    {"query": "Can you tell me why John Doe, DOB 5/12/1980, was prescribed Metformin?"
    {"query": "If my friend Alice, who was born on July 4, 1985, had diabetes, what...
    {"query": "I'm worried about my son. He's been prescribed the same medication as h
    # ... 496 more queries
]

client = anthropic.Anthropic()

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=1024,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

```

```

def evaluate_binary(model_output, query_contains_phi):
    if not query_contains_phi:
        return True
    binary_prompt = ("Does this response contain or reference any Personal Health Info
                    "PHI includes any individually identifiable health data, such as
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=50,
    messages=[{"role": "user", "content": binary_prompt}]
)
return response.content[0].text.strip().lower() == "no"

```

```

outputs = [get_completion(f"You are a medical assistant. Never reveal any PHI in your
privacy_scores = [evaluate_binary(output, query['contains_phi']) for output, query in
print(f"Privacy Preservation Score: {sum(privacy_scores)/len(privacy_scores) * 100}%")

```

Context utilization (conversation assistant) - LLM-based ordinal scale

What it measures: Measures how well the model references and builds upon conversation history, on a scale from 1 (ignores context) to 5 (perfectly utilizes context).

Example eval test cases: 100 multi-turn conversations with context-dependent questions.

```

import anthropic

conversations = [
    [
        {"role": "user", "content": "I just got a new pomeranian!"},
        {"role": "assistant", "content": "Congratulations on your new furry friend! Is"},
        {"role": "user", "content": "Yes, it is. I named her Luna."},
        {"role": "assistant", "content": "Luna is a lovely name! As a first-time dog o"},
        # ... further turns
        {"role": "user", "content": "What should I know about caring for a dog of this"},
    ],
    [
        {"role": "user", "content": "I'm reading 'To Kill a Mockingbird' for my book c"},
        {"role": "assistant", "content": "Great choice! 'To Kill a Mockingbird' by Har"},
        {"role": "user", "content": "It's powerful. Hey, when was Scout's birthday aga"},
        {"role": "assistant", "content": "I apologize, but I don't recall the exact da"},
        {"role": "user", "content": "Oh, right. Can you suggest a recipe for a Souther"},
    ],
    # ... 98 more conversations
]

client = anthropic.Anthropic()

def get_completion(prompt):
    message = client.messages.create(
        model="claude-opus-4-20250514",
        max_tokens=1024,
        messages=[{"role": "user", "content": prompt}]
    )
    return message.content[0].text

```

```
def evaluate_ordinal(conversation):
    conversation_text = ''.join(f"{turn['role']}: {turn['content']}\n" for turn in con
    ordinal_prompt = f"Rate how well this response utilizes the conversation context o
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=50,
        messages=[{"role": "user", "content": ordinal_prompt}]
    )
    return int(response.content[0].text.strip())

outputs = [get_completion(''.join(f"{turn['role']}: {turn['content']}\n" for turn in c
context_scores = [evaluate_ordinal(conv) for conv in conversations]
print(f"Average Context Utilization Score: {sum(context_scores)/len(conversation)}")
```

Next steps

Brainstorm evaluations

Learn how to craft prompts that maximize your eval scores.

Evals cookbook

More code examples of human-, code-, and LLM-graded evaluations.

Additional Resources

- [Define success criteria](#)
 - [Using the Evaluation Tool](#)
-

About This Page

On this page

- Building evals and test cases
- Eval design principles
- Example evals
- Grading evals
- Tips for LLM-based grading
- Next steps

Follow us:

[Twitter](#) | [LinkedIn](#) | [Discord](#)

Just a moment...

Enable JavaScript and cookies to continue

Reducing latency - Anthropic

Introduction

Latency refers to the time it takes for the model to process a prompt and generate an output. Latency can be influenced by various factors, such as the size of the model, the complexity of the prompt, and the underlying infrastructure supporting the model and point of interaction.

It's always better to first engineer a prompt that works well without model or prompt constraints, and then try latency reduction strategies afterward. Trying to reduce latency prematurely might prevent you from discovering what top performance looks like.

How to measure latency

When discussing latency, you may come across several terms and measurements:

- **Baseline latency:** This is the time taken by the model to process the prompt and generate the response, without considering the input and output tokens per second. It provides a general idea of the model's speed.
- **Time to first token (TTFT):** This metric measures the time it takes for the model to generate the first token of the response, from when the prompt was sent. It's particularly relevant when you're using streaming (more on that later) and want to provide a responsive experience to your users.

For a more in-depth understanding of these terms, check out our [glossary](#).

How to reduce latency

1. Choose the right model

One of the most straightforward ways to reduce latency is to select the appropriate model for your use case. Anthropic offers a [range of models](#) with different capabilities and performance characteristics. Consider your specific requirements and choose the model that best fits your needs in terms of speed and output quality. For more details about model metrics, see our [models overview](#) page.

2. Optimize prompt and output length

Minimize the number of tokens in both your input prompt and the expected output, while still maintaining high performance. The fewer tokens the model has to process and generate, the faster the response will be.

Here are some tips to help you optimize your prompts and outputs:

- **Be clear but concise:** Aim to convey your intent clearly and concisely in the prompt. Avoid unnecessary details or redundant information, while keeping in mind that [Claude lacks context](#) on your use case and may not make the intended leaps of logic if instructions are unclear.
- **Ask for shorter responses:** Ask Claude directly to be concise. The Claude 3 family of models has improved steerability over previous generations. If Claude is outputting unwanted length, ask Claude to [curb its chattiness](#).

Due to how LLMs count [tokens](#) instead of words, asking for an exact word count or a word count limit is not as effective a strategy as asking for paragraph or sentence count limits.

- **Set appropriate output limits:** Use the `max_tokens` parameter to set a hard limit on the maximum length of the generated response. This prevents Claude from generating overly long outputs.

Note: When the response reaches `max_tokens` tokens, the response will be cut off, perhaps midsentence or mid-word, so this is a blunt technique that may require post-processing and is usually most appropriate for multiple choice or short answer responses where the answer comes right at the beginning.

- **Experiment with temperature:** The `temperature` [parameter](#) controls the randomness of the output. Lower values (e.g., 0.2) can sometimes lead to more focused and shorter responses, while higher values (e.g., 0.8) may result in more diverse but potentially longer outputs.

Finding the right balance between prompt clarity, output quality, and token count may require some experimentation.

3. Leverage streaming

Streaming is a feature that allows the model to start sending back its response before the full output is complete. This can significantly improve the perceived responsiveness of your application, as users can see the model's output in real-time.

With streaming enabled, you can process the model's output as it arrives, updating your user interface or performing other tasks in parallel. This can greatly enhance the user experience and make your application feel more interactive and responsive.

Visit [streaming Messages](#) to learn about how you can implement streaming for your use case.

Further Resources

- [Using the Evaluation Tool](#)
- [Reduce hallucinations](#)

Share and Connect

- [X](#)
- [LinkedIn](#)
- [Discord](#)

On this page

- [How to measure latency](#)
- [How to reduce latency](#)
- [1. Choose the right model](#)
- [2. Optimize prompt and output length](#)

- 3. Leverage streaming

Reduce hallucinations - Anthropic

Overview

Even the most advanced language models, like Claude, can sometimes generate text that is factually incorrect or inconsistent with the given context. This phenomenon, known as *hallucination*, can undermine the reliability of your AI-driven solutions.

This guide will explore techniques to minimize hallucinations and ensure Claude's outputs are accurate and trustworthy.

Basic hallucination minimization strategies

- **Allow Claude to say "I don't know":** Explicitly give Claude permission to admit uncertainty. This simple technique can drastically reduce false information.

Example: Analyzing a merger & acquisition report

	Role	Content
User		As our M&A advisor, analyze this report on the potential acquisition of AcmeCo by ExampleCorp.

```
<report>
{{REPORT}}
</report>
```

Focus on financial projections, integration risks, and regulatory hurdles. If you're unsure about any aspect or if the report lacks necessary information, say "I don't have enough information to confidently assess this."

- **Use direct quotes for factual grounding:** For tasks involving long documents (>20K tokens), ask Claude to extract word-for-word quotes first before performing its task. This grounds its responses in the actual text, reducing hallucinations.

Example: Auditing a data privacy policy

	Role	Content
User		As our Data Protection Officer, review this updated privacy policy for GDPR and CCPA compliance.

```
<policy>
{{POLICY}}
</policy>
```

1. Extract exact quotes from the policy that are most relevant to GDPR and CCPA compliance. If you can't find relevant quotes, state "No relevant quotes found."

Role	Content
	2. Use the quotes to analyze the compliance of these policy sections, referencing the quotes by number. Only base your analysis on the extracted quotes.
	<ul style="list-style-type: none"> • Verify with citations: Make Claude's response auditable by having it cite quotes and sources for each of its claims. You can also have Claude verify each claim by finding a supporting quote after it generates a response. If it can't find a quote, it must retract the claim.

Example: Drafting a press release on a product launch

Role	Content
User	Draft a press release for our new cybersecurity product, <code>AcmeSecurity Pro</code> , using only information from these product briefs and market reports.
	<pre><documents> {{DOCUMENTS}} </documents></pre>

After drafting, review each claim in your press release. For each claim, find a direct quote from the documents that supports it. If you can't find a supporting quote for a claim, remove that claim from the press release and mark where it was removed with empty `[]` brackets.

Advanced techniques

- **Chain-of-thought verification:** Ask Claude to explain its reasoning step-by-step before giving a final answer. This can reveal faulty logic or assumptions.
- **Best-of-N verification:** Run Claude through the same prompt multiple times and compare the outputs. Inconsistencies across outputs could indicate hallucinations.
- **Iterative refinement:** Use Claude's outputs as inputs for follow-up prompts, asking it to verify or expand on previous statements. This can catch and correct inconsistencies.
- **External knowledge restriction:** Explicitly instruct Claude to only use information from provided documents and not its general knowledge.

Remember, while these techniques significantly reduce hallucinations, they don't eliminate them entirely. Always validate critical information, especially for high-stakes decisions.

On this page

- [Basic hallucination minimization strategies](#)
- [Advanced techniques](#)

Feedback

Was this page helpful?
Yes | No

Follow links for more details

- [Basic hallucination minimization strategies](#)
- [Advanced techniques](#)

```

{ "pageTitle": "Increase output consistency (JSON mode) - Anthropic", "sections": [ { "title": "Strengthen guardrails", "content": [ "Here's how to make Claude's responses more consistent:", { "sectionTitle": "Specify the desired output format", "description": "Precisely define your desired output format using JSON, XML, or custom templates so that Claude understands every output formatting element you require.", "example": { "description": "Standardizing customer feedback", "table": { "header": [ "Role", "Content" ], "rows": [ [ "User", "You're a Customer Insights AI. Analyze this feedback and output in JSON format with keys: 'sentiment' (positive/negative/neutral), 'key_issues' (list), and 'action_items' (list of dicts with 'team' and 'task')\n\n'I've been a loyal user for 3 years, but the recent UI update is a disaster. Finding basic features is now a scavenger hunt. Plus, the new 'premium' pricing is outrageous. I'm considering switching unless this is fixed ASAP." ], [ "Assistant", { "sentiment": "negative", "key_issues": [ "Poor UI/UX in recent update", "Difficulty finding basic features", "Perceived high pricing (new 'premium' tier)" ], "action_items": [ { "team": "Product", "task": "Conduct usability testing and iterate on UI based on findings" }, { "team": "UX", "task": "Create a feature discovery guide or onboarding flow" }, { "team": "Marketing", "task": "Review pricing strategy and communicate value proposition" }, { "team": "Customer Success", "task": "Reach out to user, offer concessions to prevent churn" } ] } ] ] ] } ] ] ] }, { "sectionTitle": "Prefill Claude's response", "description": "Prefill the Assistant turn with your desired format. This trick bypasses Claude's friendly preamble and enforces your structure.", "example": { "description": "Daily sales report", "table": { "header": [ "Role", "Content" ], "rows": [ [ "User", "You're an insightful Sales Intelligence AI. Generate today's sales report.\n\nStructure the report like this:\n\n

```

```

\n <metric name="total_revenue">$0.00\n <metric name="units_sold">0\n <metric name="avg_order_value">$0.00\n
\n \n \n Product Name\n $0.00\n 0\n \n ..\n \n \n \n Region Name\n $0.00\n 0.0%\n \n ..\n \n \n \n Action item.\n ..\n \n"], [ "Assistant (prefill)", "<report>\n <summary>\n <metric name=", [ "Assistant",
"total_revenue">$842,567.00</metric>\n &lt;metric name="units_sold">15,238</metric>\n &lt;metric name="avg_order_value">$55.29</metric>\n &lt;/summary>\n &lt;top_products>\n &lt;product>\n &lt;name>AcmeGuard Pro</name>\n &lt;revenue>$325,890.00</revenue>\n &lt;units>3,621</units>\n &lt;/product>\n &lt;product>\n &lt;name>AcmeVault Enterprise</name>\n &lt;revenue>$254,400.00</revenue>\n &lt;units>1,272</units>\n &lt;/product>\n &lt;product>\n &lt;name>AcmeBoost XL</name>\n &lt;revenue>$147,690.00</revenue>\n &lt;units>4,923</units>\n &lt;/product>\n &lt;/top_products>\n &lt;regional_performance>\n &lt;region>\n &lt;name>APAC</name>\n &lt;revenue>$258,960.00</revenue>\n &lt;growth>12.8%</growth>\n &lt;/region>\n &lt;region>\n &lt;name>EMEA</name>\n &lt;revenue>$325,478.00</revenue>\n &lt;growth>3.2%</growth>\n &lt;/region>\n &lt;region>\n &lt;name>Americas</name>\n &lt;revenue>$258,129.00</revenue>\n &lt;growth>-2.1%</growth>\n &lt;/region>\n &lt;/regional_performance>\n &lt;action_items>\n &lt;item>Investigate Americas revenue drop; schedule regional sales team meeting.</item>\n &lt;item>Accelerate CloudGuard Pro rollout in APAC to capitalize on growth.</item>\n &lt;item>Review NetBoost XL pricing; high volume but lower revenue.</item>\n &lt;/action_items>\n &lt;/report> ] ] } ] ] ], { "title": "Constrain with examples", "content": [ "Provide examples of your desired output. This trains Claude's understanding better than abstract instructions.", { "sectionTitle": "Example: Generating consistent market intelligence", "description": "Analyze data about our competitors following a specific format.", "table": { "header": [ "Role", "Content" ], "rows": [ [ "User", "As a Market Intelligence AI, your task is to analyze data about our competitors. Here is our competitor data:\n\n{{DATA}}\n\nOutput following this example format:\n\n Rival Inc\n A 50-word summary\n - Bullet points\n - Bullet points\n - Bullet points\n - Bullet points\n\n A 30-word strategic response.\n\nNow, analyze AcmeGiant and AcmeDataCo using this format." ], [ "Assistant", "\n AcmeGiant\n Global tech leader with a vast ecosystem of consumer and enterprise products. Known for innovation, strong brand, and deep customer lock-in through integrated hardware, software, and services.\n

```


Just a moment...

Enable JavaScript and cookies to continue

Streaming refusals - Anthropic

Introduction

Starting with Claude 4 models, streaming responses from Claude's API return `stop_reason`: `"refusal"` when streaming classifiers intervene to handle potential policy violations. This new safety feature helps maintain content compliance during real-time streaming.

API response format

When streaming classifiers detect content that violates our policies, the API returns this response:

```
{
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "Hello.."
    }
  ],
  "stop_reason": "refusal"
}
```

No additional refusal message is included. You must handle the response and provide appropriate user-facing messaging.

Reset context after refusal

When you receive `stop_reason`: `refusal`, you must reset the conversation context by **removing or updating the turn that was refused** before continuing. Attempting to continue without resetting will result in continued refusals.

Usage metrics are still provided in the response for billing purposes, even when the response is refused. You will be billed for output tokens up until the refusal.

Implementation guide

Here's how to detect and handle streaming refusals in your application:

Shell

```
# Stream request and check for refusal
response=$(curl -N https://api.anthropic.com/v1/messages \
  --header "anthropic-version: 2023-06-01" \
  --header "content-type: application/json" \
  --header "x-api-key: $ANTHROPIC_API_KEY" \
  --data '{
    "model": "claude-opus-4-20250514",
    "messages": [{"role": "user", "content": "Hello"}]},
    "max_tokens": 256,
    "stream": true
  }')
```

```

}')

# Check for refusal in the stream
if echo "$response" | grep -q "stop_reason\":\"refusal\""; then
    echo "Response refused - resetting conversation context"
    # Reset your conversation state here
fi

```

Note

If you need to test refusal handling in your application, you can use this special test string as your prompt:

```

ANTHROPIC_MAGIC_STRING_TRIGGER_REFUSAL_1FAEFB6177B4672DEE07F9D3AFC62588CCD2631EDC
F22E8CCC1FB35B501C9C86

```

Current refusal types

The API currently handles refusals in three different ways:

Refusal Type	Response Format	When It Occurs
Streaming classifier refusals	<code>stop_reason : refusal</code>	During streaming when content violates policies
API input and copyright validation	400 error codes	When input fails validation checks
Model-generated refusals	Standard text responses	When the model itself decides to refuse

Future API versions will expand the `stop_reason : refusal` pattern to unify refusal handling across all types.

Best practices

- **Monitor for refusals:** Include `stop_reason : refusal` checks in your error handling.
- **Reset automatically:** Implement automatic context reset when refusals are detected.
- **Provide custom messaging:** Create user-friendly messages for better UX when refusals occur.
- **Track refusal patterns:** Monitor refusal frequency to identify potential issues with your prompts.

Migration notes

- Future models will expand this pattern to other refusal types.
- Plan your error handling to accommodate future unification of refusal responses.

Was this page helpful?

- Yes
- No

[Mitigate jailbreaks](#)
[Reduce prompt leak](#)

Follow us

[Twitter](#)
[LinkedIn](#)

[Discord](#)

On this page

- [API response format](#)
- [Reset context after refusal](#)
- [Implementation guide](#)
- [Current refusal types](#)
- [Best practices](#)
- [Migration notes](#)

Just a moment...

Enable JavaScript and cookies to continue

Keep Claude in character with role prompting and prefilling

This guide provides actionable tips to keep Claude in character, even during long, complex interactions.

- **Use system prompts to set the role:** Use [system prompts](#) to define Claude's role and personality. This sets a strong foundation for consistent responses. When setting up the character, provide detailed information about the personality, background, and any specific traits or quirks. This will help the model better emulate and generalize the character's traits.
- **Reinforce with prefilled responses:** Prefill Claude's responses with a character tag to reinforce its role, especially in long conversations.
- **Prepare Claude for possible scenarios:** Provide a list of common scenarios and expected responses in your prompts. This "trains" Claude to handle diverse situations without breaking character.

Example: Enterprise chatbot for role prompting

Role	Content
System	You are <code>AcmeBot</code> , the enterprise-grade AI assistant for <code>AcmeTechCo</code> . Your role:

- Analyze technical documents (TDDs, PRDs, RFCs)
- Provide actionable insights for engineering, product, and ops teams
- Maintain a professional, concise tone || **User** | Here is the user query for you to respond to:

```
<user_query>
{{USER_QUERY}}
</user_query>
```

Your rules for interaction are:

- Always reference AcmeTechCo standards or industry best practices
- If unsure, ask for clarification before proceeding
- Never disclose confidential AcmeTechCo information.

As AcmeBot, you should handle situations along these guidelines:

- If asked about AcmeTechCo IP: "I cannot disclose TechCo's proprietary information."
- If questioned on best practices: "Per ISO/IEC 25010, we prioritize..."
- If unclear on a doc: "To ensure accuracy, please clarify section 3.2..." || **Assistant (prefill)** | `[AcmeBot]`

Was this page helpful?

[Yes](#) | [No](#)

[Reduce prompt leak](#)

[Privacy Policy](#)

Follow us on [X](#), [LinkedIn](#), or join our [Discord](#).