ChatGPT

xAI Guides Compilation

This document combines key guidance from multiple xAI documentation pages: **Reasoning**, **Chat**, **The Hitchhiker's Guide to Grok (Tutorial)** and **Tools Overview**. Each section summarizes the original content, preserves headings and code examples, and provides citations to the relevant documentation lines or screenshots.

Reasoning

The **Reasoning** guide explains how xAI's reasoning models, such as `grok-4`, think through problems and expose their internal thought processes. Variants labelled `grok-4-non-reasoning` are based on `grok-4` but disable reasoning. Parameters like `presencePenalty`, `frequencyPenalty` and `stop` are not supported by reasoning models [1].

Key Features

- **Think Before Responding:** The model reasons step-by-step before delivering an answer [2].
- **Math & Quantitative Strength:** Strong at numerical challenges and logic puzzles [3].
- **Reasoning Trace:** The model's thoughts are available via the `reasoning_content` or `encrypted_content` fields in the response object [4]. You can access this trace through the `message.reasoning_content` attribute of the chat completion response [5]. For `grok-4` the trace may be encrypted if `use_encrypted_content=true` [6].

Control How Hard the Model Thinks

The `reasoning_effort` parameter (not supported by `grok-4`) controls how much time the model spends thinking. It accepts two values [7]:

- `low` – minimal thinking time for quick responses.
- `high` – maximum thinking time for complex problems.

Usage Example

Below is a complete example showing how to use `grok-4` to multiply `101` by `3` and report both the completion and reasoning token counts. Note that reasoning models require a longer timeout to allow for their internal reasoning:

```
import os

from xai_sdk import Client
from xai_sdk.chat import system, user
```

```
client = Client(
    api_key=os.getenv("XAI_API_KEY"),
    timeout=3600,  # Override default timeout with longer timeout for reasoning
models
)

chat = client.chat.create(
    model="grok-4",
    messages=[system("You are a highly intelligent AI assistant.")],
)
chat.append(user("What is 101*3?"))

response = chat.sample()

print("Final Response:")
print(response.content)
print("\nNumber of completion tokens:")
print(response.usage.completion_tokens)
print("\nNumber of reasoning tokens:")
print(response.usage.reasoning_tokens)
```

Sample output:

```
Final Response:
The result of 101 multiplied by 3 is 303.

Number of completion tokens:
14

Number of reasoning tokens:
310
```

(Code and output from the Reasoning page [8].)

Notes on Consumption

Reasoning tokens are counted in your final consumption, so using a higher `reasoning_effort` will increase token usage [9].

Chat

The **Chat** guide introduces the xAI chat completions API. Chat accepts “text in, text out” and is the most popular feature of xAI API [10]. It supports summarizing articles, creative writing, Q&A, customer support and even coding tasks [10].

Prerequisites

- **xAI Account:** You need an xAI account to access the API [11].
- **API Key:** Ensure your API key has access to the chat endpoint and the chat model [12].

If you don't know how to create these, follow the Hitchhiker's Guide to Grok (see tutorial section) [13]. You can create an API key via the xAI console [14].

Basic Chat Completions Example

Chat requests send a list of messages to the API; the model processes them and returns a response. You can also stream the response, covered in the Streaming Response guide [15].

Here is a simple chat example using the xAI Python SDK [16]:

```
import os

from xai_sdk import Client
from xai_sdk.chat import user, system

client = Client(
    api_key=os.getenv("XAI_API_KEY"),
    timeout=3600,
)

chat = client.chat.create(model="grok-4")
chat.append(system("You are a PhD-level mathematician."))
chat.append(user("What is 2 + 2?"))

response = chat.sample()
print(response.content)
```

The response would be:

```
'2 + 2 equals 4.'
```

Conversations

xAI's API is stateless; each chat request is processed independently without automatically using previous history. To preserve context, you must include prior messages in the `messages` list [17]. Messages have roles:

- `system` – defines how the model should respond (instructions) [18].
- `user` – represents user requests or data [18].
- `assistant` – used for the model's responses or to inject previous answers [19].

For example, a conversation might look like the following (JSON format) [20]:

```
[
  {
    "role": "system",
    "content": [{ "type": "text", "text": "You are a helpful and funny
assistant." }]
  },
  {
    "role": "user",
    "content": [{ "type": "text", "text": "Why don't eggs tell jokes?" }]
  },
  {
    "role": "assistant",
    "content": [{ "type": "text", "text": "They'd crack up!" }]
  },
  {
    "role": "user",
    "content": [{ "type": "text", "text": "Can you explain the joke?" }]
  }
]
```

Message Role Order Flexibility

Unlike some providers, xAI allows mixing `system`, `user` and `assistant` messages in any order [21]. You can have multiple system messages at different points in the conversation or start with a user message first. Example structures include multiple system messages or user-first messages [22].

The Hitchhiker's Guide to Grok (Tutorial)

This tutorial walks you through getting started with the xAI API [23].

Step 1: Create an xAI Account

You need to create an xAI account to access the API [24]. After creating an account, add credits before using the API [25].

Step 2: Generate an API Key

Create an API key from the **API Keys Page** in the xAI console [26]. Store it securely, ideally as an environment variable or in a `.env` file [27].

Step 3: Make Your First Request

With your API key available, you can make your first API request. The tutorial uses `curl` to send a request to the `/chat/completions` endpoint [28]:

```
curl https://api.x.ai/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $XAI_API_KEY" \
  -m 3600 \
  -d '{
    "messages": [
      {
        "role": "system",
        "content": "You are Grok, a highly intelligent, helpful AI assistant."
      },
      {
        "role": "user",
        "content": "What is the meaning of life, the universe, and everything?"
      }
    ],
    "model": "grok-4",
    "stream": false
  }'
```

Step 4: Make a Request from Python or Javascript

You can also perform the same request via the xAI Python SDK (or compatible OpenAI/Anthropic SDKs) [29]. For Python [30]:

```
import os

from xai_sdk import Client
from xai_sdk.chat import user, system

client = Client(
    api_key=os.getenv("XAI_API_KEY"),
    timeout=3600,
)

chat = client.chat.create(
    model="grok-4",
    messages=[system("You are Grok, a highly intelligent, helpful AI
assistant.")],
)
chat.append(user("What is the meaning of life, the universe, and everything?"))
```

```
response = chat.sample()
print(response.content)
```

Step 5: Use Grok to Analyze Images

Certain Grok models accept both text and images. For example [31]:

```
import os

from xai_sdk import Client
from xai_sdk.chat import user, image

client = Client(
    api_key=os.getenv("XAI_API_KEY"),
    timeout=3600,
)

chat = client.chat.create(model="grok-4")
chat.append(
    user(
        "What's in this image?",
        image("https://science.nasa.gov/wp-content/uploads/2023/09/web-first-images-release.png")
    )
)

response = chat.sample()
print(response.content)
```

The agent will respond with a description such as:

> *"This image is a photograph of a region in space, specifically a part of the Carina Nebula, captured by the James Webb Space Telescope..."* [32].

Monitoring Usage

As you use your API key, you are charged based on the number of tokens used. You can monitor usage on the xAI Console Usage Page [33]. For per-request tracking, each API response includes a `usage` object detailing prompt (input) and completion (output) token usage [34]. An example usage object [35]:

```
{
  "usage": {
    "prompt_tokens": 37,
    "completion_tokens": 530,
    "total_tokens": 800,
```

```
    "prompt_tokens_details": {
      "text_tokens": 37,
      "audio_tokens": 0,
      "image_tokens": 0,
      "cached_tokens": 8
    },
    "completion_tokens_details": {
      "reasoning_tokens": 233,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    },
    "num_sources_used": 0
  }
}
```

If you send requests too frequently or with long prompts, you might hit rate limits; see the Consumption and Rate Limits guide for details [36].

Next Steps

After learning the basics, explore the **Models** page to start building with one of xAI's latest models [37].

Tools Overview

The **Tools Overview** page explains xAI's agentic server-side tool calling. Unlike traditional tool calling where clients handle each tool invocation, xAI's agentic API manages the entire reasoning and tool-execution loop on the server [38]. Using version 1.3.1 or higher of the xAI SDK is required to access these features [39].

Tools Pricing

Agentic requests incur costs based on **token usage** and **tool invocations** [40]. Because the agent autonomously decides how many tools to call, costs scale with query complexity. See the pricing page for more details [41].

Agentic Tool Calling

When you provide server-side tools in a request, the xAI server runs an autonomous reasoning loop. Instead of returning tool calls for the client to execute, the agent researches, analyzes and responds automatically [42]. Behind the scenes, the model follows an iterative reasoning process [43]:

1. **Analyzes the query and current context** to determine what information is needed.
2. **Decides what to do next** – either make a tool call to gather more data or provide a final answer [44].
3. **If making a tool call** – selects the appropriate tool and parameters [45].
4. **Executes the tool** in real time on the server and receives the results [46].

5. **Processes the tool response** and integrates it with previous context [47].
6. **Repeats** the reasoning loop as needed until enough information is gathered [48].
7. **Returns the final response** once sufficient information is available [49].

Core Capabilities

xAI's agentic tool calling supports several core tools [50]:

- **Web Search:** Real-time search across the internet and web browsing [51].
- **X Search:** Semantic and keyword search across X posts, users and threads [52].
- **Code Execution:** Write and execute Python code for calculations, data analysis and complex computations [53].
- **Image/Video Understanding:** Optional analysis of visual content in search results [54].

Quick Start Example

It's recommended to use the xAI Python SDK in **streaming** mode to get real-time observability and immediate feedback [55]. A quick start example to get the latest updates from xAI might look like this [56]:

```
import os

from xai_sdk import Client
from xai_sdk.chat import user
from xai_sdk.tools import web_search, x_search, code_execution

client = Client(api_key=os.getenv("XAI_API_KEY"))
chat = client.chat.create(
    model="grok-4-fast",  # reasoning model
    tools=[
        web_search(),
        x_search(),
        code_execution(),
    ],
)

# Ask a question
chat.append(user("What are the latest updates from xAI?"))

is_thinking = True
for response, chunk in chat.stream():
    # Display tool calls as they occur
    for tool_call in chunk.tool_calls:
        print(f"\nCalling tool: {tool_call.function.name} with arguments:
{tool_call.function.arguments}")
    if response.usage.reasoning_tokens and is_thinking:
        print(f"\rThinking... ({response.usage.reasoning_tokens} tokens)",
end="", flush=True)
```

```
    if chunk.content and is_thinking:
        print("\n\nFinal Response:")
        is_thinking = False
    if chunk.content and not is_thinking:
        print(chunk.content, end="", flush=True)

print("\n\nCitations:")
print(response.citations)
print("\n\nUsage:")
print(response.usage)
print(response.server_side_tool_usage)
```

This script streams the response, prints each tool call as it happens, and finally prints citations and usage statistics. During streaming you will see `Thinking...` messages and tool call notifications [56].

A synchronous (non-streaming) version waits for the full response before printing anything [57]:

```
import os

from xai_sdk import Client
from xai_sdk.chat import user
from xai_sdk.tools import web_search, x_search, code_execution

client = Client(api_key=os.getenv("XAI_API_KEY"))
chat = client.chat.create(
    model="grok-4-fast",
    tools=[
        web_search(),
        x_search(),
        code_execution(),
    ],
)
chat.append(user("What is the latest update from xAI?"))

# Wait for the entire agentic process to finish
response = chat.sample()

print("\n\nFinal Response:")
print(response.content)

print("\n\nCitations:")
print(response.citations)

print("\n\nUsage:")
print(response.usage)
print(response.server_side_tool_usage)
```

```
print("\n\nServer Side Tool Calls:")
print(response.tool_calls)
```

Understanding the Agentic Tool Calling Response

The agentic tool calling API provides rich observability into the research process. You can view real-time tool call decisions via the `tool_calls` attribute on each `chunk` during streaming [58]. Only tool call invocations are shown; the outputs of those calls are not returned directly [59]. When using the xAI SDK in streaming mode, the SDK accumulates `tool_calls` in the final `response` object for later inspection [60].

Real-time Server-side Tool Calls

To print each tool call as it happens, iterate over `chunk.tool_calls` during streaming and inspect each `function.name` and `function.arguments` [58]:

```
for tool_call in chunk.tool_calls:
    print(f"\nCalling tool: {tool_call.function.name} with arguments:
{tool_call.function.arguments}")
```

Citations

The `citations` attribute on the `response` object lists all URLs encountered during the agent's search process [61]. Citations are returned only when the agentic request completes and are not available in real time [62]. Not every URL may be relevant to the final answer because the agent may discard sources that are not useful [63].

Server-side Tool Calls vs Tool Usage

The API distinguishes between two metrics [64]:

- `tool_calls` **– All Attempted Calls:** A list of every attempted tool call, including those that fail [65]. Each entry is a `ToolCall` object containing an `id`, `function.name` and `function.arguments` [66].
- `server_side_tool_usage` **– Successful Calls (Billable):** A map of tools that executed successfully and how many times each was called [67]. Only these calls are billable [68].

Tool Call Function Names vs Usage Categories

The `tool_calls` list uses precise function names, while `server_side_tool_usage` uses higher-level categories. The mapping is [69]:

Usage Category	Function Name(s)
SERVER_SIDE_TOOL_WEB_SEARCH	`web_search`, `web_search_with_snippets`, `browse_page`
SERVER_SIDE_TOOL_X_SEARCH	`x_user_search`, `x_keyword_search`, `x_semantic_search`, `x_thread_fetch`
SERVER_SIDE_TOOL_CODE_EXECUTION	`code_execution`
SERVER_SIDE_TOOL_VIEW_X_VIDEO	`view_x_video`
SERVER_SIDE_TOOL_VIEW_IMAGE	`view_image`

When Tool Calls and Usage Differ

`tool_calls` and `server_side_tool_usage` will differ when [70]:

- **Failed tool executions:** The agent browses a non-existent web page, fetches a deleted X post or encounters other execution errors.
- **Invalid parameters:** Tool calls with malformed arguments.
- **Network or service issues:** Temporary failures in the tool execution pipeline.

The agent is robust and continues searching even after failures [71]. Only successful calls are billed [68].

Understanding Token Usage

Agentic requests have unique token usage patterns [72]:

- `completion_tokens` – tokens for the final output text [73].
- `prompt_tokens` – cumulative input tokens across all inference requests [74]. Agentic workflows involve multiple reasoning steps; however prompt caching allows shared context to be reused, improving efficiency [75].
- `reasoning_tokens` – tokens used for the model's internal reasoning, excluding final outputs [76].
- `cached_prompt_text_tokens` – number of prompt tokens served from cache [77].
- `prompt_image_tokens` – tokens from visual content (images or videos) processed [78].
- `prompt_text_tokens` **and** `total_tokens` – text tokens in prompts (excluding special tokens) and the sum of all token types [79].

Synchronous Agentic Requests (Non-streaming)

You can make non-streaming requests for simpler use cases, waiting for the entire agentic process to finish before receiving the response [80]. The synchronous example shown earlier demonstrates how to access final content, citations, usage and tool calls in one go [57].

Using Tools with OpenAI Responses API

Agentic tool calling also works via the OpenAI Responses API. For example [81]:

```python
import os

from openai import OpenAI

api_key = os.getenv("XAI_API_KEY")
client = OpenAI(
    api_key=api_key,
    base_url="https://api.x.ai/v1",
)

response = client.responses.create(
    model="grok-4-fast",
    input=[
        {
            "role": "User",
            "content": "What is the latest update from xAI?",
        },
    ],
    tools=[
        {"type": "web_search"},
        {"type": "x_search"},
    ],
)

print(response)
```

Agentic Tool Calling Requirements and Limitations

- **Model Compatibility:** Supported models are `grok-4`, `grok-4-fast` and `grok-4-fast-non-reasoning` [82]. It is strongly recommended to use `grok-4-fast` for agentic tool calling [83].
- **Tool Configuration:** Only server-side tools are allowed; you cannot mix server-side and client-side tools in the same request [84].
- **Request Constraints:** Batch requests (`n > 1`) are not supported. Structured output formats are not available yet. Only `temperature` and `top_p` sampling parameters are respected [85]. These constraints may be relaxed in future updates [86].

FAQ and Troubleshooting

If you see empty or incorrect content when using agentic tool calling with the xAI SDK, ensure you are using SDK version `1.3.1` or higher [87].

This combined document brings together key points from multiple xAI guides, preserving the original instructions, code examples and context. Use it as a single reference for reasoning models, chat completions, getting started with Grok and understanding the agentic tools ecosystem.

---

1 2 3 4 5 6 7 8 9 Reasoning

https://docs.x.ai/docs/guides/reasoning

10 11 12 13 14 15 16 17 18 19 20 21 22 Chat

https://docs.x.ai/docs/guides/chat

23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 The Hitchhiker's Guide to Grok | xAI Docs

https://docs.x.ai/docs/tutorial

38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 Overview

https://docs.x.ai/docs/guides/tools/overview