This is an excellent use case for a tool like `aicache`. Let's break down how the Gemini and Claude CLIs work and how you can "get in the middle" of them.

The key takeaway is that these CLIs generally don't have a built-in plugin system for this purpose. Therefore, the **Wrapper Script** (also known as a "shim") is the most robust and common approach.

---

1. How to Interact with the CLIs

This is where the two services differ significantly. Gemini's primary CLI is part of the official `gcloud` suite, whereas Claude does not have an official first-party CLI. Users typically interact with Claude via popular third-party tools or their own scripts.

A. Google Gemini CLI (`gcloud`)

The Gemini API is integrated into the Google Cloud SDK (`gcloud`). It's not a standalone `gemini` executable but a subcommand within the larger `gcloud` tool.

- **Invocation:**
    - It's a subcommand, typically of the form: `gcloud ai models predict ...` or the more recent `gcloud beta gemini ...`.
    - It is a standalone executable in the sense that you run it from your shell, not as a library.
    - **Example:**
    - codeBash

None

```
gcloud ai models predict \
  --model="gemini-1.0-pro-001" \
  --project="your-gcp-project" \
  --region="us-central1" \
```

    - `--json-request="request.json"`

    -
-
- **Input/Output:**
    - **Input:** Prompts and parameters are passed via command-line flags. Often, the entire request payload (which includes the prompt, temperature, etc.) is specified in a JSON file and passed with a flag like `--json-request`. You can also provide input directly from a file using a syntax like `--prompt-file=my-prompt.txt`.
    - **Output:** The response is printed to **standard output (stdout)**, typically as a JSON object. Errors are printed to **standard error (stderr)**.
-
- **Context:**
    - **Single-turn:** For a simple prompt, the context is the content of the prompt itself, passed in the JSON request.
    - **Multi-turn/History:** For conversational context, the entire history of `{"role": "user", "content": "..."}` and `{"role": "model", "content": "..."}` turns must be constructed and passed in the JSON request payload for each new call. The `gcloud` CLI itself is stateless; it doesn't automatically remember the previous turn. Your script or workflow is responsible for maintaining this history.
-

B. Anthropic Claude CLI (Third-Party Tools)

Since there isn't an official `claude` CLI, we'll analyze the most popular and well-designed open-source tool that users adopt: **by Simon Willison**. Many other tools follow similar Unix-philosophy principles.

- **Invocation:**
    - It's a standalone executable: `llm`. The model is specified with a flag.
    - **Example:**
    - codeBash

None

```
# Simple prompt via command-line argument
llm -m claude-3-opus "Explain the theory of relativity in 50 words."

# Prompt via standard input (stdin)
```

```
cat my_code.py | llm -m claude-3-sonnet "Refactor this Python code for clarity."
```

- **Input/Output:**
  - **Input:** The CLI is designed to work beautifully with shell pipelines. It accepts prompts from either a final command-line argument or from **standard input (stdin)**.
  - **Output:** The response is streamed directly to **standard output (stdout)**.
- **Context:**
  - The `llm` tool has a built-in, elegant way of handling context. It saves conversations to a local SQLite database.
  - You can continue a conversation using the `-c` or `--continue` flag.
  - **Example:**
  - codeBash

```
None
# Start a conversation
llm -m claude-3-haiku "My name is Alex."

# Continue it (llm automatically fetches the history of the last conversation)
llm -c "What is my name?"
```

  - This context is managed by `llm` itself, not by passing a large context file with each call. Your wrapper would need to account for this if you want to cache conversational turns correctly.

---

2. How to Intercept and Override

This is the core of your integration task.

Plugin/API System

Neither `gcloud` nor common third-party tools like `llm` have a plugin/hook system designed for intercepting and modifying I/O in the way you need. Their extensibility comes from being good "shell citizens" that can be scripted and wrapped.

Wrapper Scripts: The Definitive Solution

This is the ideal approach. You create a script that pretends to be the original CLI. The user calls your script, which then decides whether to call the *real* CLI or return a cached result.

How it Works:

1. **Find the real executable:** Locate the absolute path of the original CLI (e.g., `which gcloud` or `which llm`). Let's say it's `/usr/bin/gcloud`.
2. **Create your wrapper:** Create a new executable script (e.g., in `~/.local/bin/gcloud`).
3. **Prioritize your wrapper:** Ensure the directory containing your wrapper (`~/.local/bin`) comes *before* the original's directory in your shell's `$PATH` environment variable.
4. codeBash

```
None
# In your ~/.bashrc or ~/.zshrc
```

5. `export PATH="$HOME/.local/bin:$PATH"`

6.
7. **Implement the logic:** Your wrapper script receives all the command-line arguments. It then does the following:
    - Parses the arguments to extract the prompt and other key parameters (model name, temperature, etc.) that define a unique request.
    - Generates a cache key from these parameters.
    - Calls `aicache` to check if a response exists for this key.
    - **Cache Hit:** If a response is found, print it to stdout and exit successfully. Do **not** call the real CLI.
    - **Cache Miss:** If no response is found:

- Execute the *real* CLI, passing along all the original arguments (`/usr/bin/gcloud "$@"`).
- Capture its stdout (the response).
- Capture its stderr (for error handling).
- If the call was successful, use `aicache` to store the response with the generated key.
- Print the captured response to stdout so the user sees it.

8.

Example Pseudo-code Wrapper for a tool like

code Bash

```
None
#!/bin/bash

# 1. Define the path to the REAL executable
REAL_LLM_PATH=$(which -p llm) # Or hardcode it: /opt/homebrew/bin/llm

# 2. Parse arguments to build a cache key.
# This is the most complex part. You need to identify the prompt
# and any flags that change the output, like the model (-m).
# For simplicity, we'll hash all arguments. A real implementation
# would be smarter.
# We also need to handle stdin.
if [ -t 0 ]; then
  # Input is from arguments
  PROMPT_CONTENT="$*"
else
  # Input is from stdin (piped)
  PROMPT_CONTENT=$(cat) # Read from stdin
  # We need to re-pipe this to the real command later
fi

CACHE_KEY=$(echo "$PROMPT_CONTENT" | shasum -a 256)

# 3. Check the cache
CACHED_RESPONSE=$(aicache --get "$CACHE_KEY")
```

```
if [ $? -eq 0 ]; then
  # 4. CACHE HIT
  echo "--- (aicache HIT) ---" >&2
  echo "$CACHED_RESPONSE"
  exit 0
else
  # 5. CACHE MISS
  echo "--- (aicache MISS) ---" >&2

  # Execute the real command and capture its output
  # Handle both piped and argument-based input
  if [ -n "$PROMPT_CONTENT" ] && [ ! -t 0 ]; then
    # We read from stdin, so we must pipe it to the real command
    REAL_RESPONSE=$(echo "$PROMPT_CONTENT" | "$REAL_LLM_PATH" "$@")
  else
    # Arguments were passed directly
    REAL_RESPONSE=$("$REAL_LLM_PATH" "$@")
  fi

  # Capture the exit code of the real command
  EXIT_CODE=$?

  if [ $EXIT_CODE -eq 0 ]; then
    # On success, update the cache
    aicache --set "$CACHE_KEY" "$REAL_RESPONSE"
  fi

  # Print the response for the user
  echo "$REAL_RESPONSE"
  exit $EXIT_CODE
fi
```

Note: This is a simplified example. A production-ready version would need more robust argument parsing, especially for a complex tool like

---

3. Existing Implementations (for Reference)

While there may not be projects that specifically wrap Gemini/Claude *for caching*, there are many open-source tools that demonstrate the principles of wrapping CLIs or interacting with LLM APIs from the command line.

- **Simon Willison's** This is the best reference. While you would be *wrapping* it, its own source code is a masterclass in how to interact with different LLM APIs, manage settings, and handle input/output from the command line. Its plugin architecture (e.g., `llm-claude`) is a great example of modular API interaction.
- **(**: This is another excellent CLI tool that wraps LLM APIs (primarily OpenAI's). Studying its code will show you how it handles context, parses arguments, and makes API calls. It's a great example of the "user -> CLI -> API" flow you are trying to intercept.
- **Generic Git Wrappers:** Search on GitHub for "git-wrapper". You will find many examples of scripts that intercept `git` commands to add extra functionality (like logging, custom checks, etc.). The technique is identical to what you need to do: a script that sits in the `$PATH` and calls the real executable. This is a battle-tested pattern.